Faculdade de Tecnologia de São Paulo - FATEC-SP
Departamento de Tecnologia da Informação
Curso de Processamento de Dados

Implementação de um Compilador que Gera uma Representação Gráfica do Programa Compilado

Bruno de Brito Coimbra

Faculdade de Tecnologia de São Paulo - FATEC-SP
Departamento de Tecnologia da Informação
Curso de Processamento de Dados

Implementação de um Compilador que Gera uma Representação Gráfica do Programa Compilado

Bruno de Brito Coimbra

Monografia submetida como exigência parcial para a obtenção do Grau de Tecnólogo em Processamento de Dados
Orientador: Prof. Dr. Silvio do Lago Pereira

São Paulo
2011

"Wyrd bið ful arœd"

Sumário

1 Introdução **1**

2 Referencial Teórico **2**

3 Implementação **17**

4 Conclusão **27**

Bibliografia **32**

A Listagem dos códigos-fonte **34**

Lista de Figuras

1 Árvores Sintáticas Ambíguas 8

2 Tabela de Hashes 13

3 Exemplo Grafo Gerado pelo dot 16

4 Módulos do Compilador 19

5 Representação Gráfica Atual do Programa Fibonacci 28

6 Melhoria da Representação Gráfica do Programa Fibonacci 31

Listagens

1 Exemplo de Gramática Ambígua 7
2 Gramática Exemplo 9
3 Programa Exemplo 9
4 Formato Especificação YACC 10
5 Exemplo de especificação YACC 11
6 Código de 3 Endereços 14
7 Exemplo Gerador de Código 15
8 Exemplo de Grafo Expresso em DOT 16
9 Exemplo de Cálculo da Sequência de Fibonacci 17
10 Gramática reconhecida 17
11 Instrução de Escrita 21
12 Instrução de Atribuição 21
13 Função geradora do comando while em C 24
14 Programa Fibonacci Compilado em C 24
15 Programa Fatorial 25
16 Programa Fatorial Compilado em C 25
17 Grafo DOT Fibonacci 27
18 Melhoria Grafo DOT do Programa Fibonacci 29
19 compiler.c 34
20 global.h 35
21 scanner.l 35
22 parser.y 36
23 util.h 39
24 util.c 39
25 symtab.h 41
26 symtab.c 42
27 cgen.h 43
28 cgen.c 44
29 dotgen.h 46
30 dotgen.c 47

Lista de Acrônimos

ASCII American Standard Code for Information Interchange

BNF Backus-Naur Form

LALR Look-Ahead LR

RI Representação Intermediária

SLR Simple LR

YACC Yet Another Compiler Compiler

Resumo

Este trabalho tem como principal objetivo descrever a implementação de um compilador capaz de gerar uma representação gráfica da lógica do programa. Mais precisamente, este compilador gera duas representações, semanticamente equivalentes, do programa-fonte: uma em linguagem C, que pode ser compilada por um compilador C padrão, e outra em linguagem DOT que, ao ser compilada, gera uma representação gráfica da lógica do programa. Espera-se que esta representação gráfica seja uma ferramenta que facilite a aprendizagem de programação, uma vez que ela torna explícito fluxo de execução do programa para os novatos nesta área.

Palavras-chave: compiladores, Linguagem C, DOT, Lex, Yacc

Abstract

This work aims as main objective to describe the implementation of a compiler that can generate a graphical representation of program logic. Precisely, this compiler generates two program-objects: one in C language that can be compiled with a standard C compiler and another in DOT language when compiled build a graphical representation of program logic. Is hoped this graphical representation could be a tool that help people to learn programming, because it shows, explicitly, the execution program flow for novices in this knowledge branch.

Key words: compilers, C language, DOT, Lex, YACC

1 Introdução

Este trabalho tem como principal objetivo descrever a implementação de um compilador capaz de gerar uma representação gráfica da lógica do programa. Mais precisamente, este compilador gera duas representações, semanticamente equivalentes, do programa-fonte: uma em linguagem C, que pode ser compilada por um compilador C padrão, e outra em linguagem DOT que, ao ser compilada, gera uma representação gráfica da lógica do programa. Espera-se que esta representação gráfica seja uma ferramenta que facilite a aprendizagem de programação, uma vez que ela torna explícito fluxo de execução do programa para os novatos nesta área.

Serão apresentadas técnicas e exemplos de análise léxica e sintática, geração de código objeto e algumas estruturas de dados necessárias para a implementação. Também serão demonstrados neste trabalho a utilização de ferramentas de auxílio ao desenvolvimento de compiladores, como geradores de analisadores léxicos e sintáticos.

O trabalho foi estruturado de forma que o leitor faça uma leitura linear, sem que haja saltos entre as seções. Na primeira seção deste trabalho (Referencial Teórico) são apresentados, de forma sucinta, os conceitos fundamentais sobre os tema.

Na seção seguinte (Implementação) é discutida a implementação do compilador propriamente dito, quais foram as técnicas, ferramentas, algoritmos e estruturas de dados utilizadas. A última seção (Conclusão) apresenta os resultados obtidos, bem como as limitações do projeto e sugestões de melhoria.

No Apêndice encontra-se a listagem completa dos programas-fonte. O projeto completo também pode ser encontrado em `http://github.com/bbcoimbra/compiler`, mesmo local em que serão incluídas as correções e melhorias.

2 Referencial Teórico

2.1 Compiladores

Segundo Aho et al. (2008), *compilador* é um programa que traduz um programa-fonte para um programa-objeto. Se o programa-objeto for executável, então ele estará num formato que um computador possa executá-lo. Dessa forma, um compilador recebe como entrada um arquivo contendo um programa escrito em uma linguagem previamente determinada (programa-fonte) e produz como saída um programa objeto semanticamente equivalente ao programa recebido como entrada. Dessa forma, podemos dizer que um *compilador* é, também, um tradutor. Adicionalmente o *compilador* tem como tarefa reportar os erros encontrados durante o processo de tradução.

O *compilador* pode ser dividido em alguns módulos para efetuar o processo de tradução. A lista abaixo foi proposta por Louden (2004):

- Analisador Léxico;
- Analisador Sintático;
- Analisador Semântico;
- Gerador de Código.

O *Analisador Léxico* é o responsável por agrupar os caracteres, contidos no arquivo do programa-fonte, em unidades significativas, chamadas *tokens*, e encaminhá-las para o *Analisador Sintático*.

Por sua vez, o *Analisador Sintático* verifica se o fluxo de tokens recebidos pelo Analisador Léxico é válido para a gramática (ou linguagem) que foi definida. Usualmente o Analisador Sintático produz uma estrutura (tipicamente uma do tipo árvore) que representa o programa fonte (AHO et al., 2008).

O *Analisador Semântico* recebe a estrutura produzida pelo Analisador Sintático e, principalmente, verifica se as operações são coerentes para os tipos de dados utilizados e faz a inferência dos tipos de dados. Ao final do processo temos uma *Árvore Anotada*. As notas (informações de inferência, por exemplo) podem ser incluídas diretamente na estrutura recebida do Analisador Sintático, ou numa estrutura auxiliar como uma *Tabela de Símbolos*.

Estando a Árvore Anotada disponível para o *Gerador de Código*, este executa a tradução das estruturas recebidas para o programa-objeto. Nessa fase da compilação podem ser incluídas otimizações para que o programa traduzido execute de forma mais eficiente.

Podem haver outras fases intermediárias durante o processo, como, por exemplo, as fases de Otimização de Código, dependentes ou não da máquina-alvo.

Nas próximas Seções, discutiremos mais profundamente cada uma dessas etapas do processo de compilação.

2.2 Análise Léxica

O processo de Análise Léxica consiste em agrupar os caracteres do arquivo de entrada em unidades numa estrutura chamada *token*. Um token também é chamado de *lexema*.

Segundo Ferreira (1986), "lexema é o elemento que encerra o significado da palavra". Ou seja, é o menor conjunto de caracteres representativos para uma gramática de uma linguagem. Dessa forma, o Analisador Léxico remove a responsabilidade de representar os tokens do Analisador Sintático, simplificando sua implementação.

Segundo Aho et al. (2008), os tokens são definidos como segue:

> *Um token consiste em dois componentes, um nome de token e um valor de atributo. Os nomes de token são símbolos abstratos usados pelo analisador para fazer o reconhecimento sintático. Frequentemente, chamamos esses nomes de token de* ***terminais****, uma vez que eles aparecem como* ***símbolos terminais*** *na gramática para uma linguagem de programação. O valor do atributo, se houver, é um apontador para a tabela de símbolos que contém informações adicionais sobre o token. (...).*

Ainda segundo Aho et al. (2008), o Analisador Léxico possui algumas atribuições adicionais, como por exemplo, remover espaços em branco e comentários, efetuar contagem de linhas correlacionando um erro com o número da linha em que este foi encontrado.

Tipicamente, o Analisador Léxico não gera o todo o fluxo de tokens de uma vez. Em vez disso, a demanda de análise dos tokens fica sob a responsabilidade do Analisador Sintático, que recebe os tokens ativando uma função disponibilizada pelo Analisador Léxico (LOUDEN, 2004).

O reconhecimento dos tokens é feito utilizando duas técnicas principais:

Expressões Regulares e Autômatos Finitos. Trataremos das Expressões Regulares na Seção 2.2.1. Para uma introdução à Teoria dos Autômatos consulte Louden (2004) e Aho, Sethi e Ullman (1988), para um estudo mais detalhado, consulte Hopcroft, Motwani e Ullman (2001).

2.2.1 Expressões Regulares

Segundo Jargas (2001):

> *Resumidamente, uma expressão regular é um método formal de se especificar um padrão de texto.*
>
> *Mais detalhadamente, é uma composição de símbolos, caracteres com funções especiais, que, agrupados entre si e com caracteres literais, formam uma sequência, uma expressão. Essa expressão é interpretada como uma regra, que indicará sucesso se uma entrada de dados qualquer casar com essa regra, ou seja, obedecer exatamente a todas as suas condições*

As expressões regulares são uma importante notação para especificar os padrões dos lexemas. Mesmo não podendo especificar todos os padrões possíveis elas são muito eficientes para o propósito de especificar os tokens que necessitamos.

Definições (segundo Aho et al. (2008)):

Alfabeto é qualquer conjunto finito de símbolos. Temos como exemplos de símbolos as letras, dígitos etc. O conjunto $\{0, 1\}$ representa o *alfabeto binário*.

Cadeia em um alfabeto é uma sequência finita de símbolos retirados de alfabeto. Normalmente, o tamanho da cadeia s é dado por $|s|$. Por exemplo "compilador" é uma cadeia de tamanho 10. A cadeia vazia, indicada por ϵ, tem tamanho zero.

Linguagem é qualquer conjunto contável de cadeias de algum alfabeto.

Expressão Regular Básica são, simplesmente, os caracteres separados do alfabeto que casam com eles mesmos. Por exemplo, dado que definimos o conjunto dos caracteres ASCII como nosso alfabeto, a expressão regular $/a/$ casa com o caractere **a**.

Há três operações básicas utilizando Expressões Regulares conforme descritas abaixo (LOUDEN, 2004):

Escolha Entre Alternativas Dado que r e s são expressões regulares, então $r|s$ é uma expressão regular que case com a expressão r ou com a expressão s. Exemplo: dado que r seja a expressão regular $/a/$ e s a expressão regular $/b/$, $r|s$ casa com o caractere **a** ou o caractere **b**.

Concatenação A concatenação de duas expressões regulares r e s é da pela expressão rs e casa com qualquer cadeia que case com a expressão regular r seguida pela expressão regular s. Exemplo: dado que r seja a expressão regular $/ca/$ e s a expressão regular $/sa/$, rs casa com a cadeia "casa".

Repetição Também conhecida como fecho de Kleene, é denotada por $r*$, em que r é uma expressão regular. A expressão regular $r*$ representa o conjunto de cadeias obtidas pela concatenação de zero ou mais expressões regulares r. Exemplo: dada a expressão regular $/a*/$, esta expressão casa com as cadeias ϵ, **a**, **aa**, **aaa**, **aaaa**, ...

Para simplificar a notação das expressões regulares é comum associar nomes às expressões regulares longas. Uma Expressão Regular nomeada é chamadas de **definição regular**.

Além das operações descritas, a norma ISO/IEC 9945:2003 (2004) define operações adicionais chamadas Expressões Regulares Extendidas:

Uma ou Mais Repetições A repetição de de uma ou mais vezes da expressão regular r é dada por $r+$, eliminando o casamento da expressão vazia (ϵ). O mesmo resultado poderia ser obtido com a expressão $rr*$, mas esta é uma situação tão frequente que foi simplificada e padronizada (LOUDEN, 2004).

Qualquer Caractere Um "." (ponto) é utilizado para efetuar o casamento com qualquer caractere, exceto um caractere nulo.

Há outras operações que envolvem expressões regulares. Para mais informações consulte Jargas (2001), ISO/IEC 9945:2003 (2004) e Louden (2004).

2.3 Análise Sintática

A Análise Sintática define a forma com que um programa é estruturado. Essa estrutura é dada por um conjunto de *regras gramaticais* descritas em uma *Gramática Livre de Contexto* (verificar Seção 2.3.1).

Segundo Aho et al. (2008):

> *Existem três estratégias gerais de análise sintática para o processamento de gramáticas: universal, descendente e ascendente. Os métodos de análise baseados na estratégia universal (...) podem analisar qualquer gramática,*

> *(...) no entanto são muito ineficientes para serem utilizados em compiladores de produção.*
>
> *Os métodos geralmente usados em compiladores são baseados nas estratégias descendentes ou ascendentes. Conforme sugerido por seus nomes, os métodos de análise descendentes constroem as árvores de derivação de cima (raiz) para baixo (folhas), enquanto os métodos ascendentes fazem a análise no sentido inverso, começam nas folhas e avançam até a raiz construindo a árvore. Em ambas as estratégias, a entrada do analisador sintático é consumida da esquerda para a direita, um símbolo de cada vez.*

Para maiores referências sobre analisadores descendentes (descendente recursivos e LL(k)), consulte Louden (2004), Parr (2007), Jacobs (1985).

2.3.1 Gramáticas Livres de Contexto

Gramática é essencialmente um conjunto de Regras de Produção (ou Reescrita). Essas regras são, usualmente, descritas utilizando uma notação chamada **Forma de Backus-Naur**, ou **BNF** (LOUDEN, 2004).

Um exemplo abstrato de regra de produção é demonstrado abaixo:

$$A \rightarrow \alpha$$

Esta expressão indica que o não-terminal A será substituído pela sequência de terminais e/ou não-terminais representada por α. Um *terminal*, normalmente, é um *token* oriundo do analisador léxico.

Um exemplo mais concreto é demonstrado abaixo:

$$expr \rightarrow expr + expr | \textbf{numero}$$

Esta regra indica que uma expressão é composta de uma expressão seguida de um sinal de + seguida de outra expressão, ou de um número. O nome da regra é dado pela parte que está a esquerda da seta, seu corpo é dado pelo que está a direita. O sinal | indica uma escolha de alternativas no corpo da produção. Percebemos, também, que uma regra gramatical pode ter uma definição recursiva.

Segundo Louden (2004), podemos definir uma **gramática livre de contexto** mais formalmente conforme segue:

1. Um conjunto T de **terminais**.

2. Um conjunto N de **não-terminais** (disjunto de T).

3. Um conjunto P de **produções** na forma $A \rightarrow \alpha$ em que A é um elemento de N e α é um elemento de $(T \cup N)^*$ (uma sequência de terminais e não-terminais que pode ser vazia).

4. Um **símbolo inicial** S do conjunto N.

Dessa forma, o processo de reconhecimento da linguagem inicia-se derivando o símbolo inicial da gramática, substituindo repetidamente um não-terminal pelo corpo desse não terminal (AHO et al., 2008).

Assim, uma **gramática livre de contexto** é uma *gramática* conforme definido a cima, e é livre de contexto pois a parte a esquerda de uma regra de produção pode ser substituída pelo seu corpo em qualquer ponto, independentemente de onde ocorra a parte esquerda da regra (LOUDEN, 2004).

Em contrapartida, uma produção em uma gramática sensível ao contexto é demonstrada abaixo:

$$\gamma A \beta \rightarrow \gamma \alpha \beta$$

Nesta regra, A pode ser substituído por α, somente se A estiver entre os terminais γ e β.

2.3.2 Ambiguidade

Uma gramática é dita ambígua quando ela pode gerar duas árvores de derivação distintas. Consideremos a gramática da Listagem 1 e a expressão abaixo:

$$45 + 3 * 5$$

Listagem 1: Exemplo de Gramática Ambígua

```
1 exp = exp op exp
2     | (exp)
3     | numero
4     ;
5 op  = + | - | *
6     ;
```

Essa gramática possibilita a geração de duas árvores sintáticas distintas conforme demonstrado na Figura 1.

Gramáticas que possuem essa característica geram problemas para o analisador sintático, pois não permitem representar com precisão a estrutura do programa.

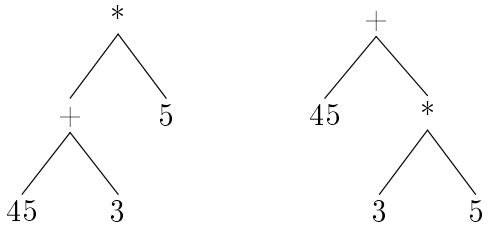

Figura 1: Árvores Sintáticas Ambíguas

A árvore a esquerda representa a expressão $(45+3)*5$ enquanto a da direita representa $45+(3*5)$.

Uma forma de resolver este problema é utilizar uma *regra de eliminação de ambiguidade*. Regras de *Precedência* e *Associatividade* são exemplos de regra de eliminação de ambiguidade. Utilizando a precedência usual da Matemática, a forma preferida para interpretar a expressão $45+3*5$ é aquela demonstrada na árvore a direita na Figura 1(LOUDEN, 2004).

2.3.3 Análise Sintática Ascendente

O processo de análise sintática ascendente foi proposto por Donald E. Knuth em 1965. Em seu artigo, ele define a análise sintática **LR(k)**. O "L" indica que a entrada é processada da esquerda para a direita e o "R" indica que uma derivação a direita é produzida, e a variável k indica o número de símbolos de verificação a frente utilizados pelo analisador. (LOUDEN, 2004).

Ainda segundo Louden (2004), essa forma de análise foi considerada impraticável até que as técnicas *SLR* e *LALR* foram desenvolvidas por DeRemer em 1969. Ainda assim, não é prática usual construir um analisador sintático ascendente manualmente, mas utilizar um gerador que abstraia os detalhes de implementação (Verificar Seção 2.3.4). Mais detalhes sobre essas técnicas podem ser obtidos em Knuth (1965), Deremer (1969), Aho et al. (2008).

Os analisadores ascendentes utilizam uma pilha explícita (diferentemente dos analisadores recursivos, que utilizam a pilha implicitamente) durante o processo de análise sintática e, no geral, possuem duas ações possíveis, além da aceitação:

1. **Carregar** um terminal da entrada para o topo da pilha.
2. **Reduzir** uma cadeia de terminais α para um não-terminal A, dada a escolha da regra $A \rightarrow \alpha$.

Por causa dessa duas ações possíveis, esse tipo de analisador também é

conhecido como **carrega-reduz** (ou *shift-reduce*).

Listagem 2: Gramática Exemplo

```
1  expressao = termo
2            ;
3
4  termo = termo + termo
5        | termo - termo
6        | fator
7        ;
8
9  fator = fator * fator
10       | (termo)
11       | numero
12       ;
```

Listagem 3: Programa Exemplo

```
1  10 * 20 + 30
```

Dados a gramática exemplo da Listagem 2 e o programa da Listagem 3, e considerando os símbolos T, F como os não-terminais *termo*, *fator* e N como o terminal *numero*, respectivamente, que *numero* representa um número inteiro, teremos os seguintes passos executados pelo analisador sintático:

1. Carregar N na pilha.
2. Reduzir o topo da pilha para F e empilhar.
3. Carregar o token "*" na pilha.
4. Carregar o token N na pilha.
5. Reduzir o topo da pilha para F e empilhar.
6. Reduzir $F * F$ para F.
7. Reduzir F para T
8. Carregar o token "+" na pilha.
9. Carregar o token N na pilha.
10. Reduzir N para F e empilhar.
11. Reduzir F para T e empilhar.
12. Reduzir $T + T$ para T.
13. Reduzir T para *expressao*.

Nesse momento, o analisador sintático retorna informando que o programa está correto, que a sequência de tokens foi aceita como válida para a gramática definida. Mais detalhes de como são feitas as escolhas entre *carregar* um token e escolher a regra para efetuar uma *redução* podem ser encontradas em Aho et al. (2008)

2.3.4 Geradores de Analisadores Sintáticos Ascendentes

Conforme citado na Seção 2.3.3, não é usual a implementação manual de um Analisador Sintático Ascendente. Dessa forma, temos alguns geradores que simplificam esse processo para o implementador de um compilador.

Um gerador amplamente utilizado é o *YACC* (do inglês *yet another compiler compiler* – *"mais um compilador de compiladores"* (LOUDEN, 2004).

Segundo Johnson (1975):

> *Yacc provides a general tool for imposing structure on the input to a computer program. The Yacc user prepares a specification of the input process; this includes rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine to do the basic input. Yacc then generates a function to control the input process. This function, called a parser, calls the user-supplied low-level input routine (the lexical analyzer) to pick up the basic items (called tokens) from the input stream. These tokens are organized according to the input structure rules, called grammar rules; when one of these rules has been recognized, then user code supplied for this rule, an action, is invoked; actions have the ability to return values and make use of the values of other actions.*

Um arquivo de especificação *YACC* possui o formato básico conforme o demonstrado na Listagem 4. Na seção de *definições*, primeira parte da especificação, é incluído entre os caracteres *%{* e *%}* os trechos de código que deverão ser incluídos diretamente no analisador sintático gerado. Normalmente, são incluídos nesse ponto da especificação os *headers* necessários, bem como as declarações de funções auxiliares necessárias.

Ainda na seção de *definições*, são incluídas as definições da união que armazenará os nós da árvore sintática produzida (verificar Apêndice A Listagem 22), dos tokens que estão presentes na gramática a ser reconhecida, o tipo de retorno dos *não-terminais* e a precedência dos operadores binários.

Listagem 4: Formato Especificação YACC

```
1 {definicoes}
2 %%
```

```
3 {regras}
4 %%
5 {rotinas auxiliares}
```

Na seção de *regras*, são definidas as regras sintáticas da gramática. Para isso é utilizada uma notação BNF. Após a definição de cada regra, é incluído entre os caracteres *{* e *}* um trecho de código em linguagem C, que representa a *ação semântica* que deverá ser executada quando aquela regra for encontrada.

Por fim, na seção de *rotinas auxiliares* são incluídas quaisquer funções que forem necessárias para a execução do analisador sintático. Comumente é incluída nessa seção a função que informa os erros encontrados pelo analisador sintático.

Listagem 5: Exemplo de especificação YACC

```
1  %{
2  #include <stdlib.h>
3  #include <stdio.h>
4
5  int yylex(void);
6  int yyerror(const char *, ...);
7
8  int resultado;
9  %}
10
11 %union{
12   int valor;
13 }
14
15 %token MAIS MENOS VEZES PARD PARE
16 %token <valor> numero
17
18 %%
19
20 expressao = termo { resultado = $1; }
21           ;
22
23 termo = termo MAIS termo  { $$ = $1 + $3 }
24       | termo MENOS termo { $$ = $1 - $3 }
25       | fator             { $$ = $1 }
26       ;
27
28 fator = fator VEZES fator { $$ = $1 * $3 }
29       | PARE termo PARD   { $$ = $2 }
30       | numero            { $$ = yylval.valor }
31       ;
32
33 %%
34
35 void yyerror(const char * s, ...) {
36   printf("erro de sintaxe: %s\n", s);
37   return;
38 }
```

Na Listagem 5 temos um exemplo concreto de uma especificação para *YACC*. Foi considerado, para essa especificação, que o analisador sintático interagiria com o analisador léxico através da chamada de função *yylex()*. Essa função retorna um inteiro que representa o tipo de token reconhecido pelo analisador léxico. Em nosso exemplo temos as representações **MAIS**, **MENOS**, **VEZES**, **PARE**, **PARD**, que significam, respectivamente os caracteres +, -, *, (,).

O token *numero* também é retornado pelo analisador léxico, mas além do valor de retorno, é disponibilizado na estrutura *yylval* o valor léxico correspondente ao token. É possível perceber na listagem 5 algumas pseudo-variáveis (aquelas iniciadas pelo caractere *$* nas ações semânticas). A variável *$$* indica o valor retornado pela ação semântica. As variáveis representadas por *$n* em que $n \in \{1, 2, 3, \dots\}$ representam os valores retornados por cada não-terminal reconhecido.

Por fim, na seção de *rotinas auxiliares*, é definida a função *yyerror()* que é ativada quando o analisador sintático encontra algum erro.

2.4 Tabela de Símbolos

Segundo Aho et al. (2008), "*Tabelas de Símbolos* são estruturas de dados utilizadas pelos compiladores para conter informações sobre as construções do programa-fonte". Essas informações são coletadas durante as fases de análise (Léxica e Sintática) e utilizadas durante a fase de geração do programa-objeto (também conhecida como fase de *síntese*).

As entradas na tabela de símbolos contém informações sobre identificadores; nome ou seu lexema, posição de memória, seu tipo, entre outras informações que o implementador julgar necessárias.

As principais operações sobre Tabelas de Símbolos são *inserir*, *consultar* e *remover* uma entrada. Dessa forma, precisamos de uma estrutura de dados que permita executar essas operações eficientemente. Foi escolhida a estrutura de *Tabela de Hash Encadeada* para sua implementação (verificar Seção 2.4.1).

Tabelas de símbolos, também, são comumente utilizadas para manter informações de escopo dos identificadores. Como neste projeto teremos apenas um escopo global, não discutiremos esse tema, entretanto, mais referências podem ser encontradas em Aho et al. (2008) e Louden (2004).

2.4.1 Hashing (Transformação de Chave)

Hashing é um método de pesquisa que utiliza uma função de transformação da chave de pesquisa para calcular o endereço em que a entrada será armazenada.

Como podemos observar na Figura 2, temos as chaves $6, 11, 16, 19, 22, 27$ inseridas na tabela. Também é possível perceber que a chave com valor 16 está inserida no endereço 0 da tabela. Para efetuar o mapeamento entre as chaves e os endereços é necessário a utilização de uma função de *hashing* (ou função de *transformação*).

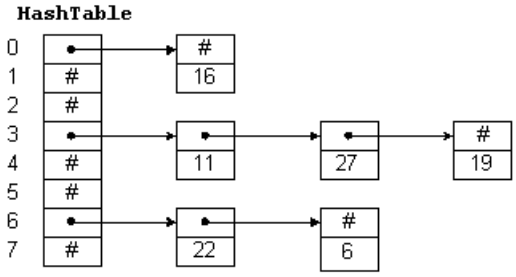

Figura 2: Tabela de Hashes

Uma **função de hashing** deve mapear uma chave (e.g. o nome de um identificador) em um inteiro dentro do intervalo $[0..M-1]$ em que M é o tamanho da tabela. Considerando que as transformações sobre as chaves são aritméticas, o primeiro passo é transformar as chaves não-numéricas em números. Para isso, podemos utilizar, por exemplo, o valor inteiro conforme a *Tabela ASCII* (ZIVIANI, 2007).

$$h(K) = K \bmod M \tag{1}$$

A Equação 1 define uma forma de transformar uma chave alfanumérica em um valor numérico K correspondente. Nela calculamos o resto da divisão de K pelo tamanho M do arranjo que armazenará a tabela.

$$K = \sum_{i=0}^{n-1} \text{chave}[i] \times i \tag{2}$$

K é definido conforme a Equação 2. i é o índice do caractere na cadeia *chave* e n é o tamanho da cadeia. O produto por i é utilizado para evitar *hashes* iguais quando tratamos de *anagramas*.

Nesse processo há grande possibilidade de duas chaves possuírem *hashes* iguais, isto é denominado *colisão* de hashes. Uma das formas possíveis de resolução de *colisões* é utilizar uma *Lista Encadeada*. Dessa forma, todas as chaves conflitantes são encadeadas em uma lista linear (ZIVIANI, 2007). Uma demonstração dessa representação é de dada na Figura 2.

Outras formas de resolução de *colisões* e outras implementações de hashes (como Hashing Perfeito) podem ser encontrados em Knuth (1973) e Ziviani (2007).

2.5 Geração de Código

Geração de Código é o processo de utilizar todas as informações geradas durante as fases de *análise* (Léxica, Sintática etc) para gerar o programa-objeto. Conforme a arquitetura do compilador, é possível incluir outras etapas intermediárias, conhecidas como *Representações Intermediárias* (RI), que visam possibilitar otimizações no programa-objeto gerado (LOUDEN, 2004).

Uma forma possível de RI é conhecida com *código-de-três-endereços*. Este formato é conhecido desta forma pois possui a seguinte forma de instruções $x = y \mathbf{op} z$, ou seja, do lado direito da atribuição possui apenas um operador binário, seus operandos e do lado esquerdo a variável que armazena o resultado da operação. Variações são permitidas para representar, por exemplo, o sinal de menos unário $x = -y$.

Listagem 6: Código de 3 Endereços

```
1 t1 = c * d
2 t2 = a + b
3 t3 = t1 + t2
4 x = t3
```

A Listagem 6 demonstra um exemplo do código-de-três-endereços para a expressão $x = a + b + c * d$. As variáveis ti para $i \in \{1, 2, 3\}$ representam variáveis temporárias criadas pelo próprio compilador.

Para este projeto, não são geradas RIs, apenas os programas-objeto em *Linguagem C*, que posteriormente podem ser compiladas por um compilador C, como o *gcc*, gerando um programa executável, e em *Linguagem DOT* possibilitando a geração de uma representação gráfica do programa. Mais referências sobre RIs e códigos-de-três-endereços são encontradas em Aho et al. (2008) e Louden (2004).

Conforme exposto, este projeto de compilador atua como um tradutor entre linguagens. Uma abordagem semelhante foi utilizada na implementação inicial da *Linguagem C++*. Este compilador traduzia programas C++ para programas C para que pudessem, posteriormente, ser compilados por um compilador C disponível. Assim, podemos considerar o processo de compilação como um processo de tradução de uma linguagem de nível mais alto para uma outra linguagem de nível mais baixo, repetindo o processo até que seja produzido um programa executável na máquina-alvo (AHO et al., 2008).

A Geração de Código também consiste num processo de linearização das estruturas de árvores disponibilizadas pelas fases anteriores, transformando, por exemplo, uma árvore sintática em um programa C, em que as instruções são escritas linearmente em um arquivo.

A Listagem 7 demonstra uma possível implementação, em pseudocódigo, de função geradora de código, tendo como base uma árvore sintática em que cada nó possui até dois filhos. Notamos que a função pode vistar a árvore em pré-ordem, em ordem e pós-ordem.

Listagem 7: Exemplo Gerador de Código

```
1  funcao geraCodigo (no_arvore T)
2  inicio
3    gerar_codigo_preparatorio(T)
4    gerar_codigo(T)
5    gerar_codigo_preparatorio_filho_esquerda(T->filho_esquerda)
6    gerar_codigo_filho_esquerda(T->filho_esquerda)
7    gerar_codigo_preparatorio_filho_direita(T->filho_direita)
8    gerar_codigo_filho_direita(T->filho_direita)
9    gerar_codigo_final(T)
10 fim
```

Com pequenas alterações no código da Listagem 7, podemos incluir mais filhos aos nós filhos à árvore T, bem como, representar a construção de quase todas as construções necessárias para produzir o programa-objeto.

2.5.1 Linguagem DOT

Segundo Ellson et al. (2003):

> *Graphviz is a collection of software for viewing and manipulating abstract graphs. It provides graph visualization for tools and web sites in domains such as software engineering, networking, databases, knowledge representation, and bio-informatics*

Um dos softwares dessa coleção é o compilador *dot*. Segundo Gansner, Koutsofios e North (2009):

> ***dot*** *draws directed graphs. It reads attributed graph text files and writes drawings, either as graph files or in a graphics format such as GIF, PNG, SVG, PDF, or PostScript.*

dot aceita como entrada um arquivo de texto expresso na Linguagem DOT (verificar `http://graphviz.org/content/dot-language`). Essa linguagem define três tipos principais de objetos: grafos, nós e arestas. O grafo principal (mais externo) pode ser direcionado (*digraph – directed graph*, ou seja, grafo direcionado), ou não-direcionado. Em um grafo principal, é possível termos um subgrafo (*subgraph*) que permite a definições de nós e arestas (GANSNER; KOUTSOFIOS; NORTH, 2009).

Um nó é criado quando o seu nome aparece pela primeira vez no arquivo. As arestas são criadas quando dois nós são ligados pelo operador de aresta ->.

Listagem 8: Exemplo de Grafo Expresso em DOT

```
1  digraph G {
2    principal     -> analise -> execucao;
3    principal     -> inicializacao;
4    principal     -> limpeza;
5    execucao      -> gera_cadeia_caracteres;
6    execucao      -> imprime_formatado;
7    inicializacao -> gera_cadeia_caracteres;
8    principal     -> imprime_formatado;
9    execucao      -> compare;
10 }
```

Na Listagem 8 temos o exemplo de um grafo escrito em DOT que após sua compilação com o comando

dot -Tpng exemplo_dot.gv > exemplo_dot.png

gerará a representação gráfica demonstrada na Figura 3.

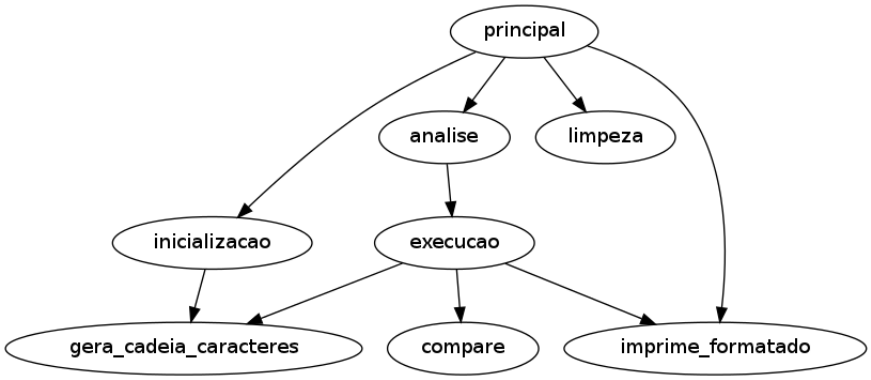

Figura 3: Exemplo Grafo Gerado pelo dot

3 Implementação

Para a implementação desse trabalho, foi definida a gramática simples que opera apenas sobre inteiros. Não ha ativação de funções e apenas um escopo (global). Na Listagem 9, temos um programa-exemplo para o cálculo da sequência de Fibonacci.

Listagem 9: Exemplo de Cálculo da Sequência de Fibonacci

```
n0 = 0;
n1 = 1;
naux = 0;
i = 0;
leia n;
n = n - 1;
se (n == 0)
  escreva n;
senao
  enquanto( i < n )
    naux = n1;
    n1 = n0 + n1;
    n0 = naux;
    i = i + 1;
  fim;
  escreva n1;
fim;
```

Verificamos no exemplo da Listagem 9, que a linguagem suporta construções comumente vistas numa linguagem de programação mais sofisticada. Temos uma série de atribuições, leitura da entrada-padrão, estrutura condicional (*se-senão*), estrutura de laço (*enquanto*) e escrita para a saída-padrão. Entretanto não há suporte para vetores (*arrays*), números de ponto flutuante e cadeias de caracteres (*strings*), recursos presentes nas linguagens de programação reais.

Mesmo na ausência destes recursos, nossa linguagem possibilita o estudo da implementação e funcionamento de um compilador. A definição formal da linguagem encontra-se na Listagem 10.

Listagem 10: Gramática reconhecida

```
program : stmts
        ;

stmts : stmts stmt
      | stmt
      ;

stmt  : if_decl SEMI
      | while_decl SEMI
      | attrib_decl SEMI
      | read_decl SEMI
      | write_decl SEMI
      ;

if_decl  : IF LPAREN bool RPAREN stmts END
         | IF LPAREN bool RPAREN stmts ELSE stmts END
         ;

while_decl : WHILE LPAREN bool RPAREN stmts END
           ;

attrib_decl : ID ATTR expr
```

```
23              ;
24
25   read_decl : READ ID
26             ;
27
28   write_decl : WRITE ID
29              ;
30
31   expr : expr PLUS expr
32        | expr MINUS expr
33        | expr TIMES expr
34        | expr OVER expr
35        | factor
36        | bool
37        ;
38   bool : expr OR expr
39        | expr AND expr
40        | expr EQ expr
41        | expr NEQ expr
42        | expr GT expr
43        | expr LT expr
44        | expr GE expr
45        | expr LE expr
46        | expr
47        ;
48
49   factor : LPAREN expr RPAREN
50          | ID
51          | NUM
52          ;
```

Como podemos perceber um programa é um conjunto de instruções delimitados por ponto-e-vírgula. A linguagem disponibiliza uma instrução para execução condicional e uma outra para laços de repetição. Ambas tomam como parâmetro uma expressão que possa ser avalidada como um inteiro. Como ocorre na *Linguagem C* uma expressão cujo valor seja avaliado como 0 (zero) é considerada *falsa* e qualquer outro valor é avaliado como *verdadeiro*.

Adicionalmente as instruções de laço e condicionais, a linguagem propicia uma instrução para o armazenamento da avaliação de uma expressão (atribuição de variável) e duas instruções de Entrada e Saída. A instrução de entrada-padrão (*leia*) lê um inteiro do teclado e a instrução de saída (*escreva*) escreve o valor atribuído a uma variável na saída-padrão (possivelmente um terminal).

As operações relacionais (booleanas) possuem prioridades conforme a Tabela 1, sendo que as operações que aparecem primeiro, na tabela, possuem maior prioridade. As operações matemáticas possuem a precedência usual.

$<$ $<=$ $>$ $>=$	Menor, Menor ou igual, Maior, Maior ou igual
$==$ $!=$	Igualdade, Desigualdade

Tabela 1: Prioridades dos Operadores Relacionais

A gramática da Listagem 10 é ambígua (verificar Seção 2.3.2), todavia as ambiguidades são resolvida pelas prioridades já discutidas, durante o processo de análise sintática.

Para a implementação do compilador foi escolhida a *linguagem C*. A escolha foi baseada na familiaridade do autor com a linguagem, além da farta disponibilidade de ferramentas e literatura sobre a referida linguagem. Mais informações podem ser obtidas em Banahan, Brady e Doran (1991), Kernighan e Ritchie (1999) e Schildt (1995)

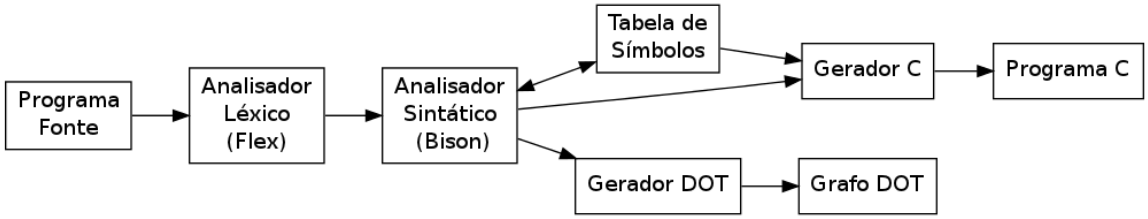

Figura 4: Módulos do Compilador

A Figura 4 demonstra o esquema de interação entre os módulos do compilador. O programa-fonte é lido pelo *Analisador Léxico* que produz o *fluxo de tokens* que alimenta o *Analisador Sintático*. Este verifica se os tokens recebidos são corretos para a *gramática* determinada, produz a *Árvore Sintática* e insere e consulta entradas na *Tabela de Símbolos*. O *Gerador C* navega na *Árvore Sintática* para gerar as construções correspondentes em C, consultando a *Tabela de Símbolos* quando necessário. O *Gerador DOT* monta a representação gráfica apenas com as informações disponibilizadas pela *Árvore Sintática*. Por fim, compilamos o programa C gerado, com um compilador C padrão, para obtermos o programa executável e compilamos o arquivo DOT (*Grafo DOT*) com o aplicativo **dot** (Seção 2.5.1) para obtermos a representação gráfica da lógica do programa-fonte.

3.1 Estruturas de Dados

As principais *Estruturas de Dados* estão definidas na Listagem 20 (global.h). São definidos três enumeradores que serão utilizados para diferenciar os nós da Árvore Sintática: *node_kind*, *stmt_kind* e *expr_kind*. O enumerador *node_kind* especifica se determinado nó é uma **instrução** ou **expressão**. Caso seja uma **instrução**, verificamos seu tipo no enumerador *stmt_kind* que possui como valores possíveis: *if_k*, *while_k*, *attrib_k*, *write_k*, *read_k*, para as instruções de escolha condicional, laço de repetição, atribuição de variável, escrita e leitura, respectivamente. Caso seja o nó seja uma **expressão**, verificamos o enumerador *expr_kind*, que nos possibilita os valores seguintes: *op_k*, *id_k* e *const_k*, para indicar expressões aritméticas e booleanas, a presença de um identificador ou uma constante numérica, respectivamente.

Os *tokens* para constantes numéricas e identificadores são armazenados

numa estrutura chamada *token_t*. A estrutura mantém a linha em que o token foi encontrado e uma união que armazena um inteiro (constante numérica), ou um ponteiro para uma cadeia de caracteres (identificadores). Para os demais casos, apenas o tipo do token é retornado pelo analisador léxico.

A Árvore Sintática é produzida por nós da estrutura do tipo *node_t*. A sequência de instruções é representada por uma lista ligada utilizando o apontador *next*. As instruções aninhadas (repetições e condicionais) são representadas como nós filhos. Para os nós que representam expressões o atributo da expressão é armazenado numa união conforme o tipo da expressão (operação aritmética, um identificador, ou uma constante numérica).

3.2 O Programa Principal

O programa principal, *compiler.c*, é bastante simples e está disponível no Apêndice A na Listagem 19.

Primeiramente são configuradas as variáveis de chaveamento, com base nas opções passadas na linha de comando. Para isso são feitas sucessivas chamadas a função *getopt()*, presente na biblioteca-padrão da linguagem C.

São configuradas as variáveis que armazenam as referências para o arquivo de entrada (que será utilizada pelo Analisador Léxico) e para a Tabela de Símbolos (utilizada no Analisador Sintático). Em seguida, é chamada a função que ativa o Analisador Sintático (*yyparse()*).

Ativações adicionais como imprimir árvore sintática e tabela de símbolos, gerar código C (Seção 3.6.1) e DOT (3.6.2), são feitas conforme as opções passadas na linha de comando.

3.3 Análise Sintática

O Analisador Sintático foi produzido utilizado o código gerado pela ferramenta *GNU/Bison* (CORBETT; STALLMAN, 2008). *Bison* é uma implementação do *YACC* (Yet Another Compiler Compiler – Um Outro Compilador de Compiladores), um gerador de analisadores sintáticos ascendentes que recebe um arquivo de especificações da gramática a ser reconhecida (descrito numa gramática livre de contexto) e produz código C para cada regra sintática reconhecida.

O arquivo da especificação está listado no Apêndice A na Listagem 22,

arquivo parser.y. A especificação é dividida em 3 partes, separadas pelos caracteres *%%*. Na primeira, são incluídas o preâmbulo e as declarações de configuração do *Bison*. Na segunda, estão as regras gramaticais e suas respectivas regras semânticas. Na terceira, inclui-se quaisquer funções auxiliares que forem necessárias.

Na seção inicial da especificação, o código listado entre *%{* e *%}* será incluído diretamente no arquivo gerado, em seguida é definido, como uma união, o tipo de dados retornado pelo analisador gerado, bem como a declaração dos tokens (que nesse contexto também são os símbolos terminais da gramática) e as precedências das operações ambíguas.

As regras gramaticais, na segunda seção do arquivo de especificação, são idênticas àquelas listadas no início deste capítulo (Seção 3), acrescidas das ações semânticas equivalentes. O exemplo da gramática para a instrução de escrita de um inteiro encontra-se na Listagem 11. Este trecho de código indica que o Analisador Sintático deve receber um token **WRITE** seguido de um token **ID** do Analisador Léxico. Quando isso ocorrer é criado um novo nó de *instrução* do tipo *escrita*, que será armazenado na pseudo-variável **$$** para ser retornado e incluído na Árvore Sintática. Também são armazenados no nó o nome do Identificador (variável) que deverá ser escrita e a sua localização no arquivo-fonte.

Listagem 11: Instrução de Escrita

```
1 write_decl : WRITE ID
2            {
3             $$ = new_stmt_node(write_k);
4             $$->child[0] = new_expr_node(id_k);
5             $$->child[0]->attr.name = copy_str ((yylval.token)->value.name);
6             $$->lineno = yylval.token->lineno;
7            }
8            ;
```

As demais regras são bastantes parecidas com aquela demonstrada na Listagem 11, exceto pela regra que define a atribuição de uma variável. Para as atribuições (verificar Listagem 12) é necessário declarar uma regra implícita para armazenarmos os valores do token **ID** antes de construirmos o nó correspondente. Isso é necessário, pois o Analisador Sintático só sabe que se trata de uma instrução de atribuição depois de reconhecer o não-terminal *expr*. Quando *expr* é reconhecido as referências para o token **ID** já foram perdidas. Outro detalhe que chama a atenção na Listagem 12 é a presença da pseudo-variável **$4**. O Bison nomeia cada terminal e não-terminal de uma produção com o caractere $ seguido com um número n em que n é o índice do terminal ou não-terminal da produção, iniciado em 1. No caso da Listagem 12 temos 3 terminais e não-terminais, sendo *expr* o terceiro, dessa forma o índice de valor 4 aparece devido a instrução implícita para armazenar, temporariamente, as referências para o token **ID**.

Listagem 12: Instrução de Atribuição

```
1  attrib_decl : ID
2              {
3                saved_name = copy_str ((yylval.token)->value.name);
4                lineno = yylval.token->lineno;
5              }
6              ATTR expr
7              {
8                $$ = new_stmt_node(attrib_k);
9                $$->child[0] = $4;
10               $$->attr.name = saved_name;
11               $$->lineno = lineno;
12               symtab_insert(stab, saved_name);
13             }
14             ;
```

Mais referências sobre o *YACC* e *Bison* podem ser encontradas em Johnson (1975), Levine, Mason e Brown (1992), Levine (2009)

3.4 Análise Léxica

O Analisador Léxico foi implementado com o auxílio da ferramenta Flex. Segundo o site do projeto (FLEX...,):

> *Flex is a tool for generating scanners. A scanner, sometimes called a tokenizer, is a program which recognizes lexical patterns in text. The flex program reads user-specified input files, or its standard input if no file names are given, for a description of a scanner to generate. The description is in the form of pairs of regular expressions and C code, called rules. Flex generates a C source file named, "lex.yy.c", which defines the function yylex(). The file "lex.yy.c"can be compiled and linked to produce an executable. When the executable is run, it analyzes its input for occurrences of text matching the regular expressions for each rule. Whenever it finds a match, it executes the corresponding C code.*

A especificação Flex para o Analisador Léxico encontra-se no Apêndice A Listagem 21. O arquivo, assim como a especificação do Analisador Sintático, é dividido em três seções separadas por um par de caracteres *%%*: definições, regras de reconhecimento e funções auxiliares.

Na primeira seção do arquivo, o segmento de código C delimitado por *%{* e *%}* é copiado diretamente para o arquivo gerado. Ainda nesta seção, são definidas as opções para a execução do Flex, bem como as definições regulares que serão utilizadas nas regras de reconhecimento dos tokens.

As regras de reconhecimento, na segunda seção do arquivo, são pares Expressões Regulares-Blocos de Código. As Expressões Regulares são instruções de casamento para os tokens, enquanto os Blocos de Código são os trechos que deverão ser

executados quanto uma determinada cadeia de caracteres do programa-fonte casar com uma das Expressões Regulares.

As possíveis ambiguidades nas regras de casamento são resolvidas com o casamento da cadeira mais longa. Na persistência da ambiguidade, a regra que foi definida primeiro ganha precedência. Caso seja encontrado algum caractere não permitido no programa-fonte uma mensagem de erro é emitida.

A seção de funções auxiliares é opcional e não foi necessária nesta implementação.

3.5 Tabela de Símbolos

Dada a simplicidade da gramática proposta (há apenas um escopo global, ...), a tabela de símbolos é necessária apenas para manter os nomes e localização, no programa-fonte, das variáveis declaradas. Não houve a necessidade de manter os tipos das variáveis, pois por definição, há apenas operações sobre números inteiros.

Sua implementação foi feita baseada numa *Tabela de Hashs de endereçamento aberto* e a Listagem está disponível no Apêndice A Listagens 25 e 26. Para mais informações, consulte a Seção 2.4.1.

As variáveis são incluídas na tabela, pelo analisador sintático, durante o reconhecimento de uma instrução de leitura ou atribuição, na primeira vez em que ela é reconhecida.

A inclusão é efetuada calculando-se *hash* do nome da variável e mapeando o valor do hash para um endereço na tabela de símbolos (que, também, é uma tabela de hashes). Caso o endereço esteja disponível, uma entrada é criada neste endereço. Caso o endereço já esteja ocupado, o conflito é resolvido com a criação de uma lista ligada, incluindo a nova entrada no final da lista.

3.6 Geração Código

A fase final no processo de compilação para nossa implementação é a Geração de Código. Para este projeto, foram escolhidas duas representações como produto do processo de compilação, uma representação em Linguagem C e outra em Linguagem DOT. Os detalhes de implementação são discutidos nas Seções 3.6.1 e 3.6.2, respectivamente. Os códigos-fonte referentes a geração de código C estão listados nas Listagens 28 e 28, os referentes a geração de código DOT, nas Listagens 29 e 30, ambas no

Apêndice A.

3.6.1 Geração Código C

O programa-objeto em Linguagem C é gerado pela função *generate_ c()* ativada pelo programa principal (Apêndice A Listagem 19). São requeridos pela função três parâmetros: o arquivo em que o programa-objeto será escrito, o ponteiro para a Árvore Sintática e o ponteiro para a Tabela de Símbolos.

A função, primeiramente, inclui no arquivo de saída os headers necessários e o início da declaração do corpo da função *main()*. Neste momento, faz-se uso da Tabela de Símbolos. Em C, as variáveis precisam ser declaradas antes de serem utilizadas, então a função *declare_ variables()* é ativada. Esta função varre a Tabela de Símbolos e inclui a declaração de todas as variáveis necessárias no arquivo de saída.

Em seguida, a função *gen_ c()* é ativada. Esta é a função mais importante desta fase da compilação. Esta função, com base no tipo de nó recebido como parâmetro, faz chamadas para as funções correspondentes que "emitem" as instruções necessárias para o arquivo de saída.

As demais funções, *emit_ while()* por exemplo (verificar Listagem 13), geram os trechos de código C correspondentes e fazem novas chamadas a *gen_ c()* para cada um dos nós filhos.

Listagem 13: Função geradora do comando while em C

```
1 void emit_while (FILE * cfile, struct node_t * node)
2 {
3   fprintf (cfile, "while (");
4   gen_c (cfile, node->child[0]);
5   fprintf (cfile, ")\n{\n");
6   gen_c (cfile, node->child[1]);
7   fprintf (cfile, "}\n");
8   return;
9 }
```

As funções que requerem Entrada e Saída (I/O) são implementadas com chamadas as chamadas *scanf()* para Entrada e *printf()* para a saída, ambas funções contidas na biblioteca-padrão da linguagem C.

Como exemplo do resultado produzido, tomemos com exemplo o programa que calcula a Sequência de Fibonacci apresentado na Seção 3 Listagem 9. Após compilado, este programa resultará no programa C apresentado na Listagem 14.

Listagem 14: Programa Fibonacci Compilado em C

```
1 #include <unistd.h>
2 #include <stdlib.h>
3 #include <stdio.h>
```

```
4  int main (int argc, char **argv)
5  {
6    int i, naux, n, n0, n1;
7  n0 = 0;
8  n1 = 1;
9  naux = 0;
10 i = 0;
11 scanf("%d", &n);
12 n = n - 1;
13 if (n == 0)
14 {
15 printf("%d", n);
16 }
17 else
18 {
19 while (i < n)
20 {
21 naux = n1;
22 n1 = n0 + n1;
23 n0 = naux;
24 i = i + 1;
25 }
26 printf("%d", n1);
27 }
28 exit(EXIT_SUCCESS)
29 }
```

Um outro exemplo (programa que calcula um número fatorial) é apresentado nas Listagens 15 e 16.

Listagem 15: Programa Fatorial

```
1 leia n;
2 produto = 1;
3 enquanto(n>1)
4   produto = produto * n;
5   n = n -1;
6 fim;
7 escreva produto;
```

Listagem 16: Programa Fatorial Compilado em C

```
1  #include <unistd.h>
2  #include <stdlib.h>
3  #include <stdio.h>
4  int main (int argc, char **argv)
5  {
6    int produto, n;
7  scanf("%d", &n);
8  produto = 1;
9  while (n > 1)
10 {
11 produto = produto * n;
12 n = n - 1;
13 }
14 printf("%d", produto);
15 exit(EXIT_SUCCESS)
16 }
```

3.6.2 Geração Código DOT

A implementação para a linguagem DOT é bastante parecida com aquela para a Linguagem C, embora seja um pouco mais complexa. É necessário manter uma variável de contexto entre as chamadas de função que geram o código DOT. Essa

variável auxilia na geração apropriada dos nós de retorno de uma instrução de repetição e o nó da próxima instrução de uma condicional.

Para isso, as chamadas de função que geram código DOT possuem um parâmetro adicional que armazena esse nó de contexto.

A função *generate_dot()* é ativada pelo programa principal (Apêndice A Listagem 19) tendo como parâmetros apenas o ponteiro para o arquivo que armazenará o programa-objeto e o ponteiro para a Árvore Sintática. A Tabela de Símbolos não é necessária, pois não há a necessidade de declaração prévia das variáveis utilizadas.

As duas funções principais para a geração do arquivo DOT são *dot_gen_graph()* e *dot_gen_shapes*. A função *dot_gen_graph()* funciona de forma semelhante a função *gen_c()* discutida na Seção 3.6.1.

A função *dot_gen_shapes()* verifica cada nó da Árvore Sintática e escreve no programa-objeto qual a forma que um nó deve assumir. Para esta implementação, apenas os nós de instrução condicional e laço de repetição possuem forma de losangolo, as demais instruções possuem o formato padrão de retângulo.

Para mais informações sobre a linguagem DOT, consulte a página, na internet, do projeto Graphviz (verificar na Bibliografia).

4 Conclusão

Acreditamos que objetivo principal deste trabalho, implementar um compilador que pudesse gerar uma representação gráfica da lógica do programa-fonte, foi cumprido. Temos uma implementação funcional de um compilador que gera como objetos um programa C e uma representação de grafo em DOT que pode ser convertido para uma imagem.

Exemplos dos resultados obtidos com o gerador de código C estão demonstrados na Seção 3.6.1 nas Listagens 14 e 16. Utilizamos o termo "exemplos de resultados" pois não é possível determinar todos os programas que serão escritos e compilados com nosso utilitário.

Os resultados obtidos com o gerador DOT não foram, exatamente, aqueles esperados (verificar discussão na Seção 4.1). A Listagem 17 demonstra o resultado atual da compilação do programa listado na Listagem 9 (Fibonacci). A Figura 5 demonstra a imagem gerada pela compilação do arquivo DOT.

Listagem 17: Grafo DOT Fibonacci

```
1  digraph program {
2  node [shape=box];
3  attrib139797576 -> attrib139797736;
4  attrib139797736 -> attrib139797896;
5  attrib139797896 -> attrib139798056;
6  attrib139798056 -> read139798144;
7  read139798144 -> attrib139798488;
8  attrib139798488 -> if139800024;
9  if139800024 -> write139798744 [label ="true"];
10 write139798744 -> write139800096;
11 if139800024 -> while139799984 [label ="false"];
12 while139799984 -> attrib139799192 [label = "true"];
13 attrib139799192 -> attrib139799496;
14 attrib139799496 -> attrib139799672;
15 attrib139799672 -> attrib139799944;
16 attrib139799944 -> while139799984;
17 while139799984 -> write139800096 [label = "false"];
18 write139800096;
19 if139800024 [shape=diamond];while139799984 [shape=diamond];}
```

Ainda que não tenhamos obtido, exatamente, o resultado esperado com a representação gráfica, acreditamos que a ferramenta concebida neste projeto ainda pode auxiliar programadores novatos em processo de aprendizagem, pois ainda que de forma limitada, é possível observar o fluxo lógico do programa na representação gráfica obtida.

O projeto possui outras limitações e possíveis melhorias para suprir essas deficiências são discutidas na Seção 4.1. Sugestões para um próximo projeto de pesquisa são expostos na Seção 4.2.

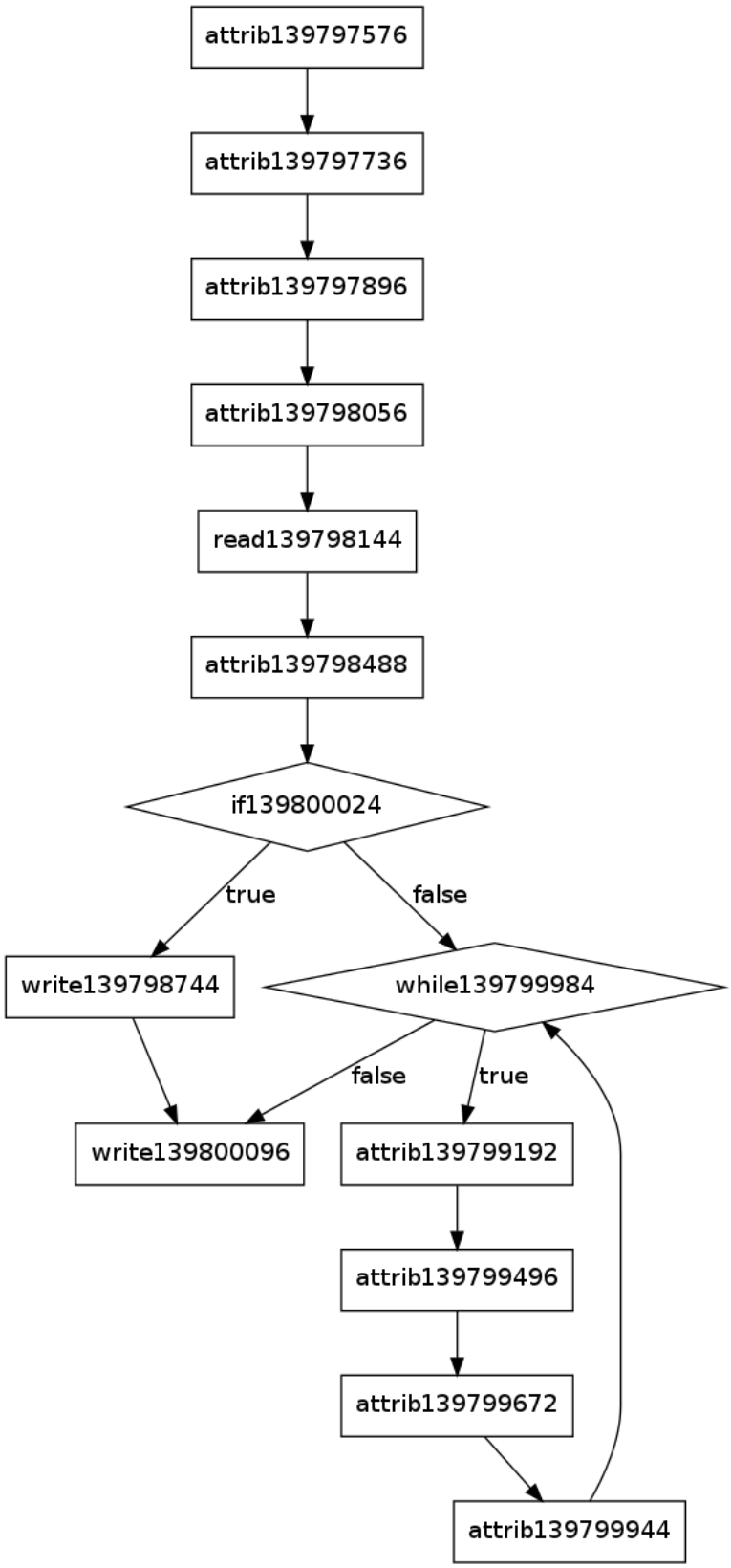

Figura 5: Representação Gráfica Atual do Programa Fibonacci

4.1 Limitações e Melhorias

A limitação principal da gramática, percebida por programadores mais experientes, é a característica da linguagem operar apenas sobre o conjunto dos números inteiros. Não há operações de ponto flutuante nem vetores. Caracteres e cadeias de caracteres (*strings*) também não são suportados. Acreditamos que programadores novatos, exatamente por serem novatos, não sejam afetados por essa percepção. Esta limitação de tipos de dados afetam apenas a geração dos programas-objetos C, pois não criam novas instruções, apenas novas expressões. Além disso, é possível operar sobre os novos tipos apenas com as instruções de atribuição, leitura, escrita, repetição e condicional já implementadas.

A inexistência de definições de procedimentos e ativação de funções é uma limitação mais severa, pois para que sejam implementadas é necessária uma alteração profunda da gramática e, por consequência, em todos os módulos do compilador.

Conforme exibido na Figura 5, percebemos que apenas as instruções são apresentadas na imagem que representa a lógica do programa-fonte. Os rótulos dos nós apresentam apenas o nome do nó, que consiste na concatenação nome do seu tipo com o endereço de memória em que ele estava alocado no momento da compilação. A melhoria consiste em executar uma passada adicional na árvore sintática para gerar os rótulos corretamente.

Um exemplo do resultado esperado é apresentado na Listagem 18. A respectiva imagem é apresentada na Figura 6

Listagem 18: Melhoria Grafo DOT do Programa Fibonacci

```
digraph program {
node [shape=box];
attrib139797576 [label = "n0 = 0"];
attrib139797736 [label = "n1 = 1"];
attrib139797896 [label = "naux = 0"];
attrib139798056 [label = "i = 0"];
read139798144 [label = "leia n"];
attrib139798488 [label = "n = n - 1"];
if139800024 [label = "n == 0"];
write139798744 [label = "escreva n"];
while139799984 [label = "i < n"];
attrib139799192 [label = "naux = n1"];
attrib139799496 [label = "n1 = n0 + n1"];
attrib139799672 [label = "n0 = naux"];
attrib139799944 [label = "i = i + 1"];
write139800096 [label = "escreva n1"];
attrib139797576 -> attrib139797736;
attrib139797736 -> attrib139797896;
attrib139797896 -> attrib139798056;
attrib139798056 -> read139798144;
read139798144 -> attrib139798488;
attrib139798488 -> if139800024;
if139800024 -> write139798744 [label = "true"];
write139798744 -> write139800096;
if139800024 -> while139799984 [label = "false"];
while139799984 -> attrib139799192 [label = "true"];
attrib139799192 -> attrib139799496;
attrib139799496 -> attrib139799672;
```

```
29 attrib139799672 -> attrib139799944;
30 attrib139799944 -> while139799984;
31 while139799984 -> write139800096 [label = "false"];
32 write139800096;
33 if139800024 [shape=diamond];while139799984 [shape=diamond];}
```

4.2 Sugestões para Projetos Futuros

Com o propósito de nortear a continuidade deste projeto segue uma lista de sugestões para possíveis melhorias:

- Implementar operações para números de ponto flutuante e caracteres;
- Implementar estruturas de vetores e matrizes;
- Implementar ativação de funções;
- Corrigir os rótulos dos nós;
- Melhorar a representação gráfica, tornando-a mais parecida com um fluxograma, inclusive incluindo outros formatos para os nós.

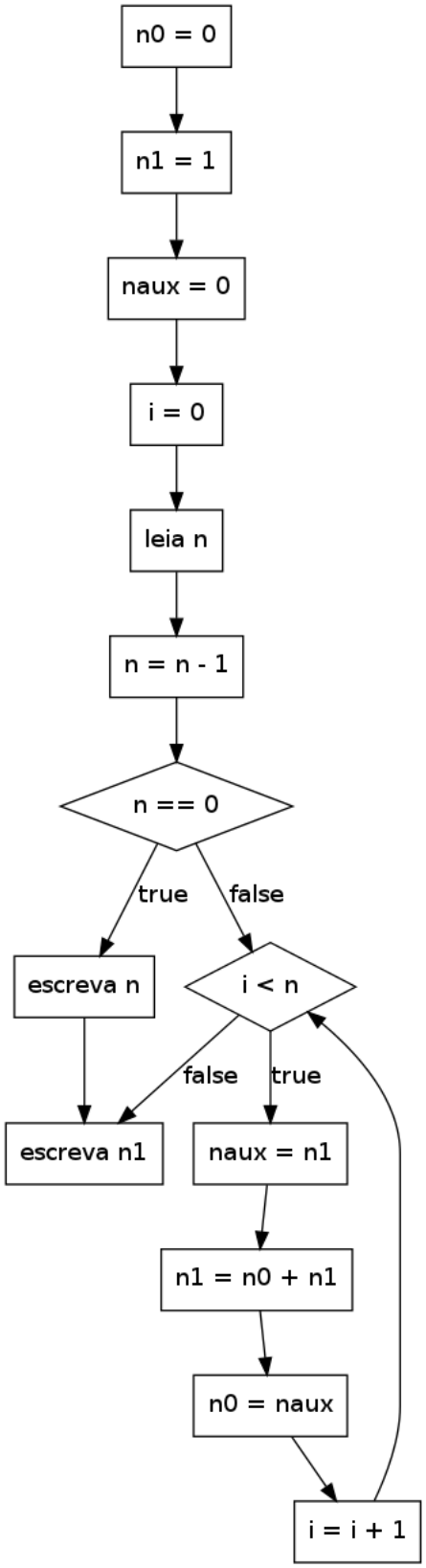

Figura 6: Melhoria da Representação Gráfica do Programa Fibonacci

Bibliografia

AHO, A. V. et al. *Compiladores: Principios, Técnicas e Ferramentas*. 2. ed. São Paulo: Pearson Addison-Wesley, 2008.

AHO, A. V.; SETHI, R.; ULLMAN, J. D. *Compilers: Principles, Techniques and Tools*. [S.l.]: Addison-Wesley, 1988.

BANAHAN, M.; BRADY, D.; DORAN, M. *The C Book*. 1991. Disponível em `http://publications.gbdirect.co.uk/c_book/`. Acessado em Maio/2011.

CORBETT, R.; STALLMAN, R. *Bison Manual*. [S.l.]: Free Software Fundation, 2008. Disponível em `http://www.gnu.org/software/bison/manual/`. Acessado em Março/2011.

DEREMER, F. L. *Practical Translators for LR(K) Languages*. Cambridge, MA, USA, 1969.

ELLSON, J. et al. *Graphviz and Dynagraph – Static and Dynamic Graph Drawing Tools*. 2003. Disponível em "`http://graphviz.org/Documentation/EGKNW03.pdf`, Acessado em Jun/2011".

FERREIRA, A. B. de H. *Novo Dicionário da Língua Portuguesa*. 2. ed. Rio de Janeiro, RJ, Brasil: Editora Nova Fronteira S.A., 1986.

FLEX Project. Disponível em `http://flex.sourceforge.net/`. Acessado em Março/2011.

GANSNER, E. R.; KOUTSOFIOS, E.; NORTH, S. *Drawing graphs with dot*. 2009. Disponível em "`http://graphviz.org/pdf/dotguide.pdf`". Acessado em Jun/2011.

HOPCROFT, J. E.; MOTWANI, R.; ULLMAN, J. D. *Introduction to automata theory, languages and computation*. Boston, Massachusetts, USA: Addison-Wesley, 2001.

ISO/IEC 9945:2003. *ISO/IEC 9945:2003: Single UNIX Specification, Version 3*. [S.l.], 2004.

JACOBS, C. Some topics in parser genaration. Vrije Universiteit, Amsterdam, 1985.

JARGAS, A. M. *Expressões Regulares - Guia de Consulta Rápida*. São Paulo, SP, Brasil: Editora Novatec, 2001.

JOHNSON, S. C. *Yacc: Yet Another Compiler-Compiler*. Murray Hill, New Jersey, USA: AT&T Bell Laboratories, 1975. Disponível em `http://dinosaur.compilertools.net/yacc/index.html`. Acessado em Abril/2011.

KERNIGHAN, B. W.; RITCHIE, D. M. *C - A Linguagem de Programacao Padrao ANSI*. Rio de Janeiro, RJ, Brasil: Elsevier Editora LTDA, 1999.

KNUTH, D. E. On the translation of languages from left to right. *Information and control*, n. 8, p. 607–639, 1965.

KNUTH, D. E. Sorting and searching. *The Art of Computer Programming*, Addison-Wesley, n. 3, 1973.

LEVINE, J. *Flex & Bison*. Sebastopol, CA, USA: O'Reilly Media Inc., 2009.

LEVINE, J. R.; MASON, T.; BROWN, D. *Lex & Yacc*. Sebastopol, CA, USA: O'Reilly Media Inc., 1992.

LOUDEN, K. C. *Compiladores: Princípios and Prática*. São Paulo: Thomson Pioneira, 2004. ISBN 8522104220.

PARR, T. *Language implementation patterns*. [S.l.]: Pragmatic Programmers, 2007.

SCHILDT, H. *C - Completo e Total*. São Paulo, SP, Brasil: Makron Books, 1995.

ZIVIANI, N. *Projeto de Algoritmos com implementação em Java e C++*. São Paulo, SP, Brasil: Thomson Learning, 2007.

A Listagem dos códigos-fonte

Listagem 19: compiler.c

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <unistd.h>
4  #include "global.h"
5  #include "util.h"
6  #include "compiler-parser.h"
7  #include "symtab.h"
8  #include "cgen.h"
9  #include "dotgen.h"
10
11 FILE *yyin;
12 extern struct symtab_t ** stab;
13
14 void usage (char * arg);
15
16 int main(int argc, char **argv, char **env)
17 {
18
19   FILE *cfile = NULL, *dotfile = NULL;
20   int print_st = 0, print_ast = 0;
21   int opt;
22
23   while ((opt = getopt(argc, argv, "std:c:")) != -1)
24   {
25     switch(opt)
26     {
27       case 's':
28         print_st = 1;
29         break;
30       case 't':
31         print_ast = 1;
32         break;
33       case 'd':
34         dotfile = fopen(optarg, "w");
35         break;
36       case 'c':
37         cfile = fopen(optarg, "w");
38         break;
39       default:
40         usage(argv[0]);
41         exit(EXIT_FAILURE);
42     }
43   }
44
45   if (optind >= argc)
46   {
47     fprintf (stderr, "Error: missed  input file.\n");
48     usage(argv[0]);
49     exit(EXIT_FAILURE);
50   }
51
52   yyin = fopen(argv[optind], "r");
53   stab = symtab_new();
54   if (yyparse() != 0)
55   {
56     fprintf(stderr, "Compilation failure!\n");
57     exit(EXIT_FAILURE);
58   }
59   if (print_ast)
60     print_tree(ast);
61   if (print_st)
62     symtab_print(stab);
63   if(cfile)
64     generate_c (cfile, ast, stab);
65   if (dotfile)
66     generate_dot (dotfile, ast);
67
68   return EXIT_SUCCESS;
69
70 }
71
72 void usage (char * arg)
```

```
73 {
74   fprintf(stderr, "Usage: %s [-s] [-t] [-d <DOT_file_name>] [-c <C_file_name>] <input_file>\n", arg);
75   return;
76 }
```

Listagem 20: global.h

```
1  #ifndef _GLOBAL_H_
2  #define _GLOBAL_H_
3
4  #include "symtab.h"
5
6  #define MAX_CHILDREN 3
7
8  enum node_kind {stmt_k, expr_k};
9  enum stmt_kind {if_k, while_k, attrib_k, write_k, read_k};
10 enum expr_kind {op_k, id_k, const_k};
11
12 struct token_t {
13   int lineno;
14   union {
15     int val;
16     char *name;
17   } value;
18 };
19
20 struct node_t {
21   struct node_t *child[MAX_CHILDREN];
22   struct node_t *next;
23   int lineno;
24   enum node_kind kind;
25   union {
26     enum stmt_kind stmt;
27     enum expr_kind expr;
28   } type;
29   union {
30     int op;
31     int val;
32     char *name;
33   } attr;
34 };
35
36 struct node_t * ast;
37 struct symtab_t ** stab;
38
39 #endif /* _GLOBAL_H_ */
```

Listagem 21: scanner.l

```
1  %{
2  #include "compiler-parser.h"
3  #include <string.h>
4  #include "global.h"
5  #include "util.h"
6  extern YYSTYPE yylval;
7  %}
8
9  %option 8bit
10 %option warn nodefault
11 %option yylineno noyywrap
12
13 ws  [[:blank:]\n]*
14 identifier  [[:alpha:]][[:alnum:]]*
15 number  0|[1-9][0-9]*
16
17 %%
18
19 "e"           return AND;
20 "ou"          return OR;
21 "se"          return IF;
22 "senao"       return ELSE;
23 "enquanto"    return WHILE;
24 "leia"        return READ;
25 "escreva"     return WRITE;
```

```
"fim"          return END;
"=="           return EQ;
"!="           return NEQ;
"="            return ATTR;
">="           return GE;
">"            return GT;
"<="           return LE;
"<"            return LT;
"+"            return PLUS;
"-"            return MINUS;
"*"            return TIMES;
"/"            return OVER;
";"            return SEMI;
"("            return LPAREN;
")"            return RPAREN;
{number}       {
                 yylval.token = new_token();
                 (yylval.token)->value.val = atoi(yytext);
                 return NUM;
               }
{identifier}   {
                 yylval.token = new_token();
                 (yylval.token)->value.name = copy_str (yytext);
                 (yylval.token)->lineno = yylineno;
                 return ID;
               }
{ws}           /* ignore */;

.              printf("bad input character '%s' at line %d\n", yytext, yylineno);
```

Listagem 22: parser.y

```
%{
#include <stdlib.h>
#include <stdio.h>
#include "global.h"
#include "util.h"
#include "symtab.h"

int yylex(void);
void yyerror(const char *, ...);

int lineno;
char * saved_name;
%}

%union{
struct token_t *token;
struct node_t *node;
}

%token AND ATTR ELSE END EQ GE GT IF LE LPAREN LT ;
%token MINUS NEQ OR OVER PLUS READ RPAREN SEMI TIMES ;
%token WHILE WRITE;
%token <val> NUM;
%token <name> ID;

%type <node> stmts stmt while_decl if_decl attrib_decl write_decl read_decl;
%type <node> bool expr factor;

%left EQ NEQ;
%left GE GT LE LT;
%left TIMES OVER;
%left PLUS MINUS;
%left LPAREN;
%nonassoc ATTR;

%%

program : stmts { ast = $1; }
        ;

stmts : stmts stmt
       {
         struct node_t *t = $1;
         if (t != NULL)
```

```
45          {
46            while (t->next != NULL)
47              t = t->next;
48            t->next = $2;
49            $$ = $1;
50          }
51          else
52           $$ = $2;
53        }
54       | stmt
55         { $$ = $1; }
56       ;
57
58 stmt  : if_decl SEMI
59         { $$ = $1; }
60       | while_decl SEMI
61         { $$ = $1; }
62       | attrib_decl SEMI
63         { $$ = $1; }
64       | read_decl SEMI
65         { $$ = $1; }
66       | write_decl SEMI
67         { $$ = $1; }
68       ;
69
70 if_decl  : IF LPAREN bool RPAREN stmts END
71            {
72              $$ = new_stmt_node(if_k);
73              $$->child[0] = $3;
74              $$->child[1] = $5;
75            }
76          | IF LPAREN bool RPAREN stmts ELSE stmts END
77            {
78              $$ = new_stmt_node(if_k);
79              $$->child[0] = $3;
80              $$->child[1] = $5;
81              $$->child[2] = $7;
82            }
83          ;
84
85 while_decl : WHILE LPAREN bool RPAREN stmts END
86            {
87              $$ = new_stmt_node(while_k);
88              $$->child[0] = $3;
89              $$->child[1] = $5;
90            }
91            ;
92
93 attrib_decl : ID
94             {
95               saved_name = copy_str ((yylval.token)->value.name);
96               lineno = yylval.token->lineno;
97             }
98             ATTR expr
99             {
100              $$ = new_stmt_node(attrib_k);
101              $$->child[0] = $4;
102              $$->attr.name = saved_name;
103              $$->lineno = lineno;
104              symtab_insert(stab, saved_name);
105            }
106            ;
107
108 read_decl : READ ID
109           {
110             $$ = new_stmt_node(read_k);
111             $$->child[0] = new_expr_node(id_k);
112             $$->child[0]->attr.name = copy_str ((yylval.token)->value.name);
113             $$->lineno = yylval.token->lineno;
114             symtab_insert(stab, (yylval.token)->value.name);
115           }
116           ;
117
118 write_decl : WRITE ID
119            {
120             $$ = new_stmt_node(write_k);
121             $$->child[0] = new_expr_node(id_k);
122             $$->child[0]->attr.name = copy_str ((yylval.token)->value.name);
```

```
123             $$->lineno = yylval.token->lineno;
124            }
125            ;
126
127 expr : expr PLUS expr
128      {
129        $$ = new_expr_node(op_k);
130        $$->child[0] = $1;
131        $$->child[1] = $3;
132        $$->attr.op = PLUS;
133      }
134      | expr MINUS expr
135      {
136        $$ = new_expr_node(op_k);
137        $$->child[0] = $1;
138        $$->child[1] = $3;
139        $$->attr.op = MINUS;
140      }
141      | expr TIMES expr
142      {
143        $$ = new_expr_node(op_k);
144        $$->child[0] = $1;
145        $$->child[1] = $3;
146        $$->attr.op = TIMES;
147      }
148      | expr OVER expr
149      {
150        $$ = new_expr_node(op_k);
151        $$->child[0] = $1;
152        $$->child[1] = $3;
153        $$->attr.op = OVER;
154      }
155      | factor
156        { $$ = $1; }
157      | bool
158        { $$ = $1; }
159      ;
160
161 bool : expr OR expr
162        {
163           $$ = new_expr_node(op_k);
164           $$->child[0] = $1;
165           $$->child[1] = $3;
166           $$->attr.op  = OR;
167        }
168      | expr AND expr
169        {
170           $$ = new_expr_node(op_k);
171           $$->child[0] = $1;
172           $$->child[1] = $3;
173           $$->attr.op  = AND;
174        }
175      | expr EQ expr
176        {
177           $$ = new_expr_node(op_k);
178           $$->child[0] = $1;
179           $$->child[1] = $3;
180           $$->attr.op  = EQ;
181        }
182      | expr NEQ expr
183        {
184           $$ = new_expr_node(op_k);
185           $$->child[0] = $1;
186           $$->child[1] = $3;
187           $$->attr.op  = NEQ;
188        }
189      | expr GT expr
190        {
191           $$ = new_expr_node(op_k);
192           $$->child[0] = $1;
193           $$->child[1] = $3;
194           $$->attr.op  = GT;
195        }
196      | expr LT expr
197        {
198           $$ = new_expr_node(op_k);
199           $$->child[0] = $1;
200           $$->child[1] = $3;
```

```
201           $$->attr.op  = LT;
202         }
203       | expr GE expr
204         {
205           $$ = new_expr_node(op_k);
206           $$->child[0] = $1;
207           $$->child[1] = $3;
208           $$->attr.op  = GE;
209         }
210       | expr LE expr
211         {
212           $$ = new_expr_node(op_k);
213           $$->child[0] = $1;
214           $$->child[1] = $3;
215           $$->attr.op  = LE;
216         }
217       | expr
218         { $$ = $1; }
219       ;
220
221 factor : LPAREN expr RPAREN
222         { $$ = $2; }
223         | ID
224           {
225             struct symtab_t * symbol = NULL;
226             $$ = new_expr_node(id_k);
227             $$->attr.name = copy_str ((yylval.token)->value.name);
228             $$->lineno = yylval.token->lineno;
229             symbol = symtab_lookup(stab, (yylval.token)->value.name);
230             if ( symbol == NULL )
231             {
232               fprintf(stderr,"Syntax error: Symbol '%s' not found, line %d.\n", (yylval.token)->value.name, yylval.
                      token->lineno);
233               exit(EXIT_FAILURE);
234             }
235           }
236         | NUM
237           {
238             $$ = new_expr_node(const_k);
239             $$->attr.val = (yylval.token)->value.val;
240           }
241         ;
242 %%
243
244 void yyerror(const char * s, ...) {
245   printf("syntax error: %s\n", s);
246   return;
247 }
```

Listagem 23: util.h

```
1  #ifndef _UTIL_H_
2  #define _UTIL_H_
3
4  struct node_t * new_expr_node (enum expr_kind kind);
5  struct node_t * new_stmt_node (enum stmt_kind kind);
6  struct token_t * new_token(void);
7  void print_node (struct node_t * node);
8  void print_tree (struct node_t * n);
9
10 char *copy_str(char * str);
11
12 #endif  /* _UTIL_H_ */
```

Listagem 24: util.c

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <string.h>
4  #include "global.h"
5  #include "util.h"
6  #include "compiler-parser.h"
7
8  static int indent_level = 0;
```

```
 9
10 struct node_t * new_expr_node(enum expr_kind kind)
11 {
12   struct node_t * n = (struct node_t *) malloc (sizeof (struct node_t));
13   int i;
14
15   for(i=0 ; i<3; i++)
16     n->child[i] = NULL;
17   n->next = NULL;
18   n->lineno = 0;
19   n->kind = expr_k;
20   n->type.expr = kind;
21   n->attr.name = NULL;
22   return n;
23 }
24
25 struct node_t * new_stmt_node(enum stmt_kind kind)
26 {
27   struct node_t *n = (struct node_t *) malloc(sizeof(struct node_t));
28   int i;
29
30   for(i=0 ; i<3; i++)
31     n->child[i] = NULL;
32   n->next = NULL;
33   n->lineno = 0;
34   n->kind = stmt_k;
35   n->type.stmt = kind;
36   return n;
37 }
38
39 struct token_t * new_token(void)
40 {
41   struct token_t * t = (struct token_t *) malloc (sizeof (struct token_t));
42   return t;
43 }
44
45 void print_node(struct node_t * node)
46 {
47   int i;
48
49   for (i = 0 ; i < indent_level ; i++)
50     fprintf(stderr, "  ");
51
52   if (node == NULL)
53     return;
54   switch (node->kind)
55   {
56     case stmt_k:
57       fprintf(stderr, "Stament: ");
58       switch (node->type.stmt)
59       {
60         case if_k:
61           fprintf(stderr, "IF\n"); break;
62         case write_k:
63           fprintf(stderr, "WRITE\n"); break;
64         case attrib_k:
65           fprintf(stderr, "ATTRIB to %s\n", node->attr.name); break;
66         case while_k:
67           fprintf(stderr, "WHILE\n"); break;
68         case read_k:
69           fprintf(stderr, "READ\n"); break;
70       }
71       break;
72     case expr_k:
73       fprintf(stderr, "Expression: ");
74       switch (node->type.expr)
75       {
76         case id_k:
77           fprintf(stderr, "ID: %s\n", node->attr.name); break;
78         case op_k:
79           fprintf(stderr, "OP: ");
80           switch (node->attr.op)
81           {
82             case AND:
83               fprintf(stderr, "&&"); break;
84             case OR:
85               fprintf(stderr, "||"); break;
86             case EQ:
```

```
87                fprintf(stderr, "=="); break;
88              case NEQ:
89                fprintf(stderr, "!="); break;
90              case GE:
91                fprintf(stderr, ">="); break;
92              case GT:
93                fprintf(stderr, ">"); break;
94              case LE:
95                fprintf(stderr, "<="); break;
96              case LT:
97                fprintf(stderr, "<"); break;
98              case PLUS:
99                fprintf(stderr, "+"); break;
100             case MINUS:
101               fprintf(stderr, "-"); break;
102             case TIMES:
103               fprintf(stderr, "*"); break;
104             case OVER:
105               fprintf(stderr, "/"); break;
106           }
107           fprintf(stderr, "\n");
108           break;
109         case const_k:
110           fprintf(stderr, "NUM: %d\n", node->attr.val); break;
111       }
112       break;
113   }
114   return;
115 }
116
117 char * copy_str (char * str)
118 {
119   char * aux;
120   int len = 0;
121   len = strlen(str) + 1;
122   aux = (char *) malloc (len);
123   memset(aux, 0, len);
124   memcpy(aux, str, len - 1);
125   return aux;
126 }
127
128 void print_tree (struct node_t * node)
129 {
130   static int first = 1;
131   int i = 0;
132
133   if (first){
134     fprintf (stderr, "\n======== Printing AST ========\n");
135     first = 0;
136   }
137
138   if (node != NULL)
139   {
140     print_node(node);
141     for (i = 0 ; i < MAX_CHILDREN ; i++)
142       if (node->child[i])
143       {
144         indent_level++;
145         print_tree(node->child[i]);
146         indent_level--;
147       }
148     if (node->next != NULL)
149       print_tree(node->next);
150   }
151   return;
152 }
```

Listagem 25: symtab.h

```
1 #ifndef _SYMTAB_H
2 #define _SYMTAB_H
3
4 #define HASH_TABLE_SIZE 1117
5 #define VAR_LOCATIONS 512
6
7 struct locations {
```

```
8   int locations[VAR_LOCATIONS + 1];
9 };
10
11 struct symtab_t {
12   char * id;
13   struct locations l;
14   struct symtab_t * next;
15 };
16
17
18 unsigned int hash (char * id);
19 struct symtab_t ** symtab_new(void);
20 int symtab_insert (struct symtab_t ** tab, char * id);
21 struct symtab_t * symtab_lookup (struct symtab_t ** tab, char * id);
22 int symtab_destroy(struct symtab_t ** tab);
23 void symtab_print(struct symtab_t ** tab);
24
25
26 #endif /* _SYMTAB_H */
```

Listagem 26: symtab.c

```
1  #include "symtab.h"
2  #include <stdlib.h>
3  #include <stdio.h>
4  #include <string.h>
5
6  unsigned int weight_array[HASH_TABLE_SIZE + 1];
7
8  void generate_weights(void)
9  {
10   int i;
11   for (i =0 ; i < HASH_TABLE_SIZE ; i++ )
12     weight_array[i] = rand();
13   weight_array[HASH_TABLE_SIZE + 1] = 0;
14   return;
15 }
16
17 unsigned int hash(char * id)
18 {
19   int len, i, j=0;
20   unsigned int h, temp=0;
21
22   len = strlen(id);
23   for (i = 0; i < len; i++)
24   {
25     if (j > (HASH_TABLE_SIZE - 1)) j = 0;
26     temp = temp + id[i] * weight_array[j];
27     j++;
28   }
29   h = temp % HASH_TABLE_SIZE;
30   return h;
31 }
32
33 struct symtab_t ** symtab_new(void)
34 {
35   int i;
36   struct symtab_t ** tab;
37   generate_weights();
38   tab = (struct symtab_t **) malloc (sizeof (struct symtab_t *) * (HASH_TABLE_SIZE + 1));
39   *(tab + HASH_TABLE_SIZE + 1) = NULL;
40   for (i = 0; i < HASH_TABLE_SIZE ; i++)
41   {
42     *(tab + i) = (struct symtab_t *) malloc (sizeof (struct symtab_t));
43     (**(tab + i)).id = NULL;
44   }
45   return tab;
46 }
47
48 int symtab_insert (struct symtab_t ** tab, char * id)
49 {
50   struct symtab_t * entry;
51   unsigned int h;
52   if (id == NULL)
53     return -1;
54   if ((entry = symtab_lookup (tab, id)) == NULL)
```

```
55    {
56      h = hash (id);
57      if ( (**(tab + h)).id == 0x0)
58      {
59        (**(tab + h)).id = strdup(id);
60      }
61      else
62      {
63        entry = *(tab + h);
64        while (entry->next != NULL)
65          entry = entry->next;
66        entry->next = (struct symtab_t *) malloc (sizeof (struct symtab_t));
67        entry->next->id = strdup (id);
68      }
69      return 1;
70    }
71    return 0;
72  }
73
74  struct symtab_t * symtab_lookup (struct symtab_t ** tab, char * id)
75  {
76    struct symtab_t * entry;
77    unsigned int h;
78
79    if (id == NULL)
80      return NULL;
81    h = hash(id);
82    if ((((**(tab+h)).id) && (strcmp ( (**(tab + h)).id, id) == 0))
83      return *(tab + h);
84    else
85    {
86      entry = *(tab + h);
87      while (entry->next != NULL)
88      {
89        entry = entry->next;
90        if (strcmp (entry->id, id) == 0)
91          return entry;
92      }
93    }
94    return NULL;
95  }
96
97  int symtab_destroy (struct symtab_t ** tab)
98  {
99    return 0;
100 }
101
102 void symtab_print(struct symtab_t ** tab)
103 {
104   int i;
105   struct symtab_t * entry;
106
107   fprintf(stderr, "\n======== Printing Symtab ========\n");
108   for (i = 0; i < HASH_TABLE_SIZE; i++)
109   {
110     entry = *(tab + i);
111     if (entry->id && (strlen(entry->id) != 0))
112     {
113       fprintf(stderr, "Entry %d:\n", i);
114       fprintf(stderr, "IDs: %s", entry->id);
115       while (entry->next != NULL)
116       {
117         entry = entry->next;
118         fprintf(stderr, ", %s", entry->id);
119       }
120       fprintf(stderr, "\n");
121     }
122   }
123   return;
124 }
```

Listagem 27: cgen.h

```
1  #ifndef _CGEN_H
2  #define _CGEN_H
3
```

```
4 void generate_c (FILE * cfile, struct node_t * ast, struct symtab_t ** stab);
5
6 #endif /* _CGEN_H */
```

Listagem 28: cgen.c

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include "global.h"
4  #include "compiler-parser.h"
5  #include "symtab.h"
6  #include "cgen.h"
7
8  void declare_variables (FILE * cfile, struct symtab_t ** stab);
9  void gen_c (FILE * cfile, struct node_t * ast);
10 void emit_op (FILE * cfile, struct node_t * node);
11 void emit_const (FILE * cfile, struct node_t * node);
12 void emit_id (FILE * cfile, struct node_t * node);
13 void emit_if (FILE * cfile, struct node_t * node);
14 void emit_while (FILE * cfile, struct node_t * node);
15 void emit_attrib (FILE * cfile, struct node_t * node);
16 void emit_read (FILE * cfile, struct node_t * node);
17 void emit_write (FILE * cfile, struct node_t * node);
18
19 void generate_c (FILE * cfile, struct node_t * ast, struct symtab_t ** stab)
20 {
21   fprintf (cfile, "#include <unistd.h>\n");
22   fprintf (cfile, "#include <stdlib.h>\n");
23   fprintf (cfile, "#include <stdio.h>\n");
24   fprintf (cfile, "int main (int argc, char **argv)\n");
25   fprintf (cfile, "{\n");
26   declare_variables (cfile, stab);
27   gen_c (cfile, ast);
28   fprintf (cfile, "exit(EXIT_SUCCESS)\n");
29   fprintf (cfile, "}\n");
30   return;
31 }
32
33 void declare_variables (FILE * cfile, struct symtab_t ** stab)
34 {
35   int i;
36   struct symtab_t * entry;
37
38   fprintf (cfile, "  int");
39   for (i = 0; i < HASH_TABLE_SIZE ; i++)
40   {
41     static int first = 1;
42     entry = *(stab + i);
43     if (entry != NULL && entry->id != NULL)
44       if (first)
45       {
46         fprintf (cfile, " %s", entry->id);
47         first = 0;
48       }
49       else
50         fprintf (cfile, ", %s", entry->id);
51   }
52   fprintf (cfile, ";\n");
53   return;
54 }
55
56 void gen_c (FILE * cfile, struct node_t * ast)
57 {
58   struct node_t * node;
59   node = ast;
60   if (node != NULL)
61   {
62     switch (node->kind)
63     {
64       case stmt_k:
65         switch (node->type.stmt)
66         {
67           case if_k:
68             emit_if (cfile, node);
69             break;
70           case while_k:
```

```
71              emit_while (cfile, node);
72              break;
73            case attrib_k:
74              emit_attrib (cfile, node);
75              break;
76            case read_k:
77              emit_read (cfile, node);
78              break;
79            case write_k:
80              emit_write (cfile, node);
81              break;
82          }
83          break;
84        case expr_k:
85          switch (node->type.expr)
86          {
87            case op_k:
88              emit_op (cfile, node);
89              break;
90            case id_k:
91              emit_id (cfile, node);
92              break;
93            case const_k:
94              emit_const (cfile, node);
95              break;
96          }
97          break;
98        default:
99          fprintf (stderr, "This shouldn't happended. You probabily found BUG.");
100         exit (EXIT_FAILURE);
101     }
102     gen_c(cfile, node->next);
103   }
104 }
105
106 void emit_op (FILE * cfile, struct node_t * node)
107 {
108   gen_c (cfile, node->child[0]);
109   switch (node->attr.op)
110   {
111     case AND:
112       fprintf (cfile, " && ");
113       break;
114     case OR:
115       fprintf (cfile, " || ");
116       break;
117     case EQ:
118       fprintf (cfile, " == ");
119       break;
120     case NEQ:
121       fprintf (cfile, " != ");
122       break;
123     case GE:
124       fprintf (cfile, " >= ");
125       break;
126     case GT:
127       fprintf (cfile, " > ");
128       break;
129     case LE:
130       fprintf (cfile, " <= ");
131       break;
132     case LT:
133       fprintf (cfile, " < ");
134       break;
135     case PLUS:
136       fprintf (cfile, " + ");
137       break;
138     case MINUS:
139       fprintf (cfile, " - ");
140       break;
141     case TIMES:
142       fprintf (cfile, " * ");
143       break;
144     case OVER:
145       fprintf (cfile, " / ");
146       break;
147   }
148   gen_c (cfile, node->child[1]);
```

```
149   return;
150 }
151 void emit_id (FILE * cfile, struct node_t * node)
152 {
153   fprintf (cfile, "%s", node->attr.name);
154   return;
155 }
156
157 void emit_const (FILE * cfile, struct node_t * node)
158 {
159   fprintf (cfile, "%d", node->attr.val);
160   return;
161 }
162
163 void emit_if (FILE * cfile, struct node_t * node)
164 {
165   fprintf (cfile, "if (");
166   gen_c (cfile, node->child[0]);
167   fprintf (cfile, ")\n{\n");
168   gen_c (cfile, node->child[1]);
169   fprintf (cfile, "}\n");
170   if (node->child[2])
171   {
172     fprintf (cfile, "else\n");
173     fprintf (cfile, "{\n");
174     gen_c (cfile, node->child[2]);
175     fprintf (cfile, "}\n");
176   }
177   return;
178 }
179
180 void emit_while (FILE * cfile, struct node_t * node)
181 {
182   fprintf (cfile, "while (");
183   gen_c (cfile, node->child[0]);
184   fprintf (cfile, ")\n{\n");
185   gen_c (cfile, node->child[1]);
186   fprintf (cfile, "}\n");
187   return;
188 }
189
190 void emit_attrib (FILE * cfile, struct node_t * node)
191 {
192   fprintf (cfile, "%s = ", node->attr.name);
193   gen_c (cfile, node->child[0]);
194   fprintf (cfile, ";\n");
195 }
196
197 void emit_read (FILE * cfile, struct node_t * node)
198 {
199   fprintf (cfile, "scanf(\"%%d\", &");
200   gen_c (cfile, node->child[0]);
201   fprintf (cfile, ");\n");
202   return;
203 }
204
205 void emit_write (FILE * cfile, struct node_t * node)
206 {
207   fprintf (cfile, "printf(\"%%d\", ");
208   gen_c (cfile, node->child[0]);
209   fprintf (cfile, ");\n");
210   return;
211 }
```

Listagem 29: dotgen.h

```
1 #ifndef _DOTGEN_H_
2 #define _DOTGEN_H_
3
4 #include <stdio.h>
5 #include "global.h"
6
7 void generate_dot (FILE * file, struct node_t * ast);
8
9 #endif /* _DOTGEN_H_ */
```

Listagem 30: dotgen.c

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <string.h>
4  #include "global.h"
5  #include "dotgen.h"
6  #include "compiler-parser.h"
7
8  char * gen_name(struct node_t *node);
9  void dot_emit_preamble (FILE * file);
10 void dot_gen_labels (FILE * file, struct node_t * node);
11 void dot_gen_graph (FILE * file, struct node_t * node, struct node_t * context);
12 void dot_gen_shapes (FILE * file, struct node_t * node);
13 void dot_emit_finally (FILE * file);
14 void dot_emit_if (FILE * file, struct node_t * node,  struct node_t * context);
15 void dot_emit_while (FILE * file, struct node_t * node, struct node_t * context);
16
17 void generate_dot (FILE * file, struct node_t * ast)
18 {
19   dot_emit_preamble(file);
20   dot_gen_labels (file, ast);
21   dot_gen_graph (file, ast, NULL);
22   dot_gen_shapes (file, ast);
23   dot_emit_finally (file);
24 }
25
26 void dot_emit_preamble (FILE * file)
27 {
28   fprintf (file, "digraph program {\n");
29   fprintf (file, "node [shape=box];\n");
30   return;
31 }
32
33 void dot_emit_finally (FILE * file)
34 {
35   fprintf (file, "}\n");
36   return;
37 }
38
39 void dot_print_op(FILE * file, int kind)
40 {
41   switch (kind)
42   {
43     case PLUS:
44       fprintf(file, " + ");
45       break;
46     case MINUS:
47       fprintf (file, " - ");
48       break;
49     case TIMES:
50       fprintf (file, " * ");
51       break;
52     case OVER:
53       fprintf (file, " / ");
54       break;
55     case EQ:
56       fprintf (file, " == ");
57       break;
58     case NEQ:
59       fprintf (file, " != ");
60       break;
61     case AND:
62       fprintf (file, " && ");
63       break;
64     case OR:
65       fprintf (file, " || ");
66       break;
67     case GE:
68       fprintf (file, " >= ");
69       break;
70     case GT:
71       fprintf (file, " > ");
72       break;
73     case LE:
74       fprintf (file, " <= ");
75       break;
76     case LT:
```

```
77        fprintf (file, " < ");
78        break;
79      default:
80        fprintf(file, "   ");
81        break;
82    }
83    return;
84  }
85  void dot_gen_if_label(FILE * file, struct node_t * node)
86  {
87    char * name;
88    if (node && node->kind == stmt_k && node->type.stmt == if_k)
89    {
90      name = gen_name (node);
91      fprintf (file, "%s [label = \" ", name);
92      dot_gen_labels (file, node->child[0]);
93      fprintf (file, "\" ];\n");
94      free (name);
95
96      dot_gen_labels (file, node->child[1]);
97      dot_gen_labels (file, node->child[2]);
98    }
99    return;
100 }
101
102 void dot_gen_while_label(FILE * file, struct node_t * node)
103 {
104   char * name;
105   if (node && node->kind == stmt_k && node->type.stmt == while_k)
106   {
107     name = gen_name (node);
108     fprintf (file, "%s [label = \" ", name);
109     dot_gen_labels (file, node->child[0]);
110     fprintf (file, "\" ];\n");
111     free (name);
112
113     dot_gen_labels (file, node->child[1]);
114   }
115   return;
116 }
117
118 void dot_gen_labels (FILE * file, struct node_t * node)
119 {
120   char *name;
121
122   if (node)
123   {
124     switch (node->kind)
125     {
126       case stmt_k:
127         switch (node->type.stmt)
128         {
129           case if_k:
130             dot_gen_if_label(file, node);
131             break;
132           case while_k:
133             dot_gen_while_label(file, node);
134             break;
135           case attrib_k:
136             name = gen_name (node);
137             fprintf (file, "%s [label = \"", name);
138             fprintf (file, "%s = ", node->attr.name);
139             dot_gen_labels (file, node->child[0]);
140             fprintf (file, "\" ];\n");
141             free (name);
142             break;
143           case read_k:
144             name = gen_name (node);
145             fprintf (file, "%s [label = \"", name);
146             fprintf(file, "leia ");
147             dot_gen_labels (file, node->child[0]);
148             fprintf (file, "\" ];\n");
149             free (name);
150             break;
151           case write_k:
152             name = gen_name (node);
153             fprintf (file, "%s [label = \"", name);
154             fprintf(file, "escreva ");
```

```
155             dot_gen_labels (file, node->child[0]);
156             fprintf (file, "\" ];\n");
157             free (name);
158             break;
159         }
160         break;
161       case expr_k:
162         switch (node->type.expr) {
163           case const_k:
164             fprintf (file, "%d", node->attr.val);
165             break;
166           case op_k:
167             dot_gen_labels (file, node->child[0]);
168             dot_print_op(file, node->attr.op);
169             dot_gen_labels (file, node->child[1]);
170             break;
171           case id_k:
172             fprintf (file, "%s", node->attr.name);
173             break;
174         }
175         break;
176     }
177     dot_gen_labels (file, node->next);
178   }
179 }
180
181 void dot_gen_graph (FILE * file, struct node_t * node, struct node_t * context)
182 {
183   char * name, * next;
184   if (node && node->kind == stmt_k)
185   {
186     switch (node->type.stmt)
187     {
188       case if_k:
189         dot_emit_if (file, node, context);
190         break;
191       case while_k:
192         dot_emit_while (file, node, context);
193         break;
194       case read_k:
195       case write_k:
196       case attrib_k:
197         name = gen_name (node);
198         fprintf (file, "%s", name);
199         if (node->next != NULL)
200         {
201           next = gen_name (node->next);
202           fprintf (file, " -> %s", next);
203           free (next);
204         }
205         else
206         {
207           if (context != NULL)
208           {
209             next = gen_name (context);
210             fprintf (file, " -> %s", next);
211             free (next);
212           }
213         }
214         fprintf(file, ";\n");
215         free (name);
216         dot_gen_graph (file, node->next, context);
217         break;
218       default:
219         fprintf(stderr, "BUG\n");
220         exit(EXIT_FAILURE);
221     }
222   }
223   return;
224 }
225
226 void dot_gen_shapes (FILE * file, struct node_t * node)
227 {
228   if (node && node->kind == stmt_k)
229   {
230     char * name;
231     int i = 0;
232     name = gen_name (node);
```

```
233     switch (node->type.stmt)
234     {
235       case if_k:
236         fprintf (file, "%s [shape=diamond];", name);
237         break;
238       case while_k:
239         fprintf (file, "%s [shape=diamond];", name);
240         break;
241       case read_k:
242       case write_k:
243       case attrib_k:
244         /* do nothing */
245         break;
246       default:
247         fprintf(stderr, "BUG\n");
248         exit(EXIT_FAILURE);
249     }
250     free(name);
251     for (i = 0; i < MAX_CHILDREN; i++)
252       dot_gen_shapes (file, node->child[i]);
253     dot_gen_shapes (file, node->next);
254   }
255   return;
256 }
257
258 void dot_emit_if (FILE * file, struct node_t * node, struct node_t * context)
259 {
260   char *node_name, *child_name;
261
262   node_name = gen_name (node);
263   child_name = gen_name (node->child[1]);
264   fprintf(file, "%s -> %s [label =\"true\"];\n", node_name, child_name);
265   dot_gen_graph (file, node->child[1], (node->next != NULL ? node->next : context));
266   if (node->child[2] != NULL)
267   {
268     free (child_name);
269     child_name = gen_name (node->child[2]);
270     fprintf(file, "%s -> %s [label =\"false\"];\n", node_name, child_name);
271     dot_gen_graph (file, node->child[2], (node->next != NULL ? node->next : context));
272   }
273   else
274   {
275     char * next;
276     if (node->next != NULL) next = gen_name (node->next);
277     else next = gen_name (context);
278     fprintf (file, "%s -> %s [label =\"false\"];\n", node_name, next);
279     free (next);
280   }
281   free (node_name); free (child_name);
282   dot_gen_graph (file, node->next, context);
283   return;
284 }
285
286 void dot_emit_while (FILE * file, struct node_t * node, struct node_t * context)
287 {
288   char *node_name, *child_name;
289   char * next;
290
291   node_name = gen_name (node);
292   child_name = gen_name (node->child[1]);
293   fprintf(file, "%s -> %s [label =\"true\"];\n", node_name, child_name);
294   dot_gen_graph (file, node->child[1], node);
295   if (node->next) next = gen_name (node->next);
296   else next = gen_name (context);
297   fprintf (file, "%s -> %s [label =\"false\"];\n", node_name, next);
298   free (next);
299   free (node_name); free (child_name);
300   dot_gen_graph (file, node->next, context);
301   return;
302 }
303
304 char * gen_name(struct node_t * node)
305 {
306   char * str;
307   str = (char *) malloc (sizeof(char) * 33);
308   memset (str, 0, 33);
309   switch (node->type.stmt)
310     {
```

```
311       case if_k:
312         sprintf(str, "if%d", (int) node);
313         break;
314       case while_k:
315         sprintf(str, "while%d", (int) node);
316         break;
317       case read_k:
318         sprintf(str, "read%d", (int) node);
319         break;
320       case write_k:
321         sprintf(str, "write%d", (int) node);
322         break;
323       case attrib_k:
324         sprintf(str, "attrib%d", (int) node);
325         break;
326       default:
327         fprintf(stderr, "BUG\n");
328         exit(EXIT_FAILURE);
329     }
330   return str;
331 }
```