Gerador de Código Intermediário

João Victor Cabral de Melo[1]

Universidade de Brasília

Resumo This project is to show what we have done and learned in the course of compilers and the grammar of experimental language C-IPL.

Keywords: Compilers · Grammar · Lexical Analyser · Flex · Syntax Analyser · Bison · Semantic Analyser · Intermediate Code Generator

1 Introdução

1.1 Motivação

A maior motivação para a implementação de um compilador é conseguir gerar código executável em máquinas, dada uma linguagem de alto nível como entrada.

1.2 Nova Primitiva

A motivação para entrada da nova primitiva *list* é a possibilidade de resolver problemas onde se precisa de uma estrutura de dados com um tamanho não definido previamente. Facilitando esse conjunto específico de problemas pela não implementação da estrutura de dados manualmente.

2 Análise Léxica

A função da leitura do compilador é identificar lexemas na linguagem e passar uma sequência desses *tokens* para o sintático avaliar se a ordem está correta.

Neste projeto foi utilizado o **flex** [Pro12] para gerar o analisador léxico, além disso foram feitas tais modificações no arquivo, como as variáveis para contagem de linhas, colunas e erros léxicos. Com isso também foi feito um tratamento de caso em que não consiga abrir o arquivo a ser escaneado, para ser entrada do usuário os lexemas a serem lidos.

A variável de escopo esta sendo tratada no léxico como uma lista sendo que cada elemento aponta para um superior. Usando está lógica e com duas variáveis globais uma para escopo atual e outra para escopo global toda vez que encontra um abre escopo andamos na lista e se fechamos um escopo voltamos para escopo superior.

A estrutura em que os lexemas são transformados é da forma linha, coluna escopo e valor. Para cada lexema lido é alocado como folha para o sintático, e também é colocado em uma lista de folhas para um futura desalocação desses lexemas lidos.

3 Análise Sintática

A função desta fase é verificar se as sentenças estão de acordo com a gramática. Como referência para construção da gramática C-IPL foi utilizado o [Hec21] e [Nal13].

Neste projeto foi utilizado o **bison** [RC21] como gerador do analisador sintático, além disso foram feitas modificações no arquivo como a inclusão de variável para contar o número de erros sintáticos e uma variável para calcular o número de argumentos de uma chamada de função.

A tabela de símbolos é montada nesta fase utilizando as sentenças de declaração de variáveis e parâmetros de variáveis como uma forma de colocar na lista ligada da tabela.

Neste projeto toda vez que se encontra uma sentença válida é colocado o id com os parâmetros do léxico. Também são colocados valores nos símbolos importantes para o semântico como se ele é parâmetro ou não,o seu tipo, se é uma função ou não. Se ele for uma função possui dois ponteiros apontando para uma lista de tipos de argumentos da função.

Também há duas funções implementadas nesta fase uma para calcular o número de argumentos de cada símbolo e outro para colocar os tipos dos argumentos de cada função na lista de tipos de argumento presente em cada símbolo. Essas funções são chamadas toda vez que encontra as sentenças de chamada de função e de declaração de função respectivamente.

O analisador sintático cria um árvore de uma forma ascendente utilizando a opção **canonical-lr** [ALSU06]. Nesta fase também se faz a árvore sintática abstrata utilizando como base a árvore dada como saída do **bison** [RC21]. A árvore consiste das produções na gramática no apêndice :

- program
- paramlist
- variableParam
- functionParam
- functionParamsList
- call
- argList
- stmList
- ifStatement
- forStatement
- returnStatement
- inputStatement
- outputStatement
- expression
- orExpression
- andExpression
- relationalExpression
- listExpression
- arithmExpression
- arithmMulDivExpression
- ID
- unaryTerm
- immutable
- const
- listOp

Para cada uma das produções é alocado um terminal que é gerado pelo analisador sintático. Sendo que cada produção pode ter no máximo 7 filhos. Cada produção tem o tipo Node e cada terminal tem o tipo Folha declarado na união. Essas produções foram escolhidas pois são as que possuem maior número terminal Folhas que são de interesse da árvore sintática abstrata.

4 Análise Semântica

A função desta fase é verificar se a árvore abstrata está com as informações corretas para o desenvolvimento do código de três endereços.

A fase foi feita em um arquivo separado verificando a tipagem de todas as expressões, os tipos de *return* de uma função, se a chamada de função possui o número correto de parâmetros, se cada parâmetro de uma chamada está de acordo com o que foi declarada na função, se o programa possui uma função *main* e se a variável já foi ou não declarada no contexto.

A primeira abordagem feita nesta fase foi adicionar nos nós da árvore um tipo. Em seguida foi feito para cada expressão uma checagem de tipos caso os tipos fossem dos tipos da básicos da linguagem C-IPL era feito a redução ou aumento do tipo ou em alguns casos específicos de **NIL** para **int list** ou **float list**.

Essas conversão de tipos foram anotadas na árvore, junto com o escopo nos *ids* e o tipo nas produções e nos terminais. Também foi adicionado dois novos tipos, um chamado ***undefined*** para fazer a verificação de tipos nos nós pais da árvore ou quando não poderia prever o tipo que teria a produção por exemplo na produção *stmList* ou quando obtivesse um erro de um filho. E outro chamado ***list*** para representar o **NIL**.

Em seguida foi feita a verificação de tipos do retorno de uma função. Para isto foi criada uma lista de retornos já lidos. Eles são verificado na produção de declaração de função se pode ou não fazer conversão de tipo neste caso foi considerado que se sua função fosse do tipo **int list** ou **float list** seu return poderia ser **NIL**. Ao final dessa produção é limpada a lista para próxima declaração.

A verificação da contagem de parâmetros de uma função foi feita com uma variável global que conta o número de argumentos de uma chamada de função na produção *call* e compara com o número guardado no respectivo símbolo na tabela de símbolos.

Caso os número estejam iguais é feita a verificação dos tipos da chamada com uma lista de argumentos da chamada criado na contagem. Essa lista é comparada item a item com a lista de argumentos da função guardado no símbolo na tabela de símbolos. Caso seja possível a conversão de tipos e feito nesta fase, se não é lançado um erro semântico.

A verificação de um programa possui uma função *main* é feita no final da execução do analisador sintático. Procura-se na tabela de símbolos se foi colocado uma função com *id* de *main* caso não ache retorna um erro.

Além disso cada folha dentro da árvore sintática abstrata possui um escopo. Caso seja variável ou declaração de função, o valor é inicializado com valor equivalente da tabela de símbolos caso contrário a folha recebe um valor de escopo negativo.

A verificação de contexto é feito toda vez na declaração de uma variável, utilizando uma busca na tabela de símbolos em caso de haver uma variável no mesmo escopo com mesmo *id* ou em caso seja usado uma variável id é lançado um erro.

5 Gerador de Código Intermediário

A função desta fase é criar uma representação de código que seja lido pelo gerador de código no *backend* [ALSU06].

Está fase só é executada quando não há erros detectados pelos três analisadores das fases anteriores. Caso não tenha erros é criado um arquivo **.tac** no mesmo diretório do arquivo e com o mesmo nome do arquivo selecionado.

Para se criar o gerador de código intermediário é analisado a tabela símbolos e a árvore abstrata sintática. Como no caso do **.table** onde são declaradas todas as variáveis na tabela de símbolos concatenadas com seu escopo e previamente com seu tipo no arquivo **.tac**.

Para lidar com as novas primitivas foram inicializados vetores de tamanho não definido com um valor zero sendo esse zero lixo de memória. Quando se atribui um valor **NIL** ao um vetor é colocado zero nele.

Quando se concatena dois valores é feita a alocação do novo valor com **mema**. Sendo o tamanho do vetor mais um e coloca-se na frente do vetor um novo valor.

Quando se faz *tail* padrão se cria um cópia do vetor e retorna o endereço da base do vetor mais um. Quando se faz um *tail destructor* pega o endereço da base do vetor e adiciona mais a este um endereço, sendo assim o novo endereço de base do vetor. Fazendo filter e map se cria uma label e se aplica a função declarada com os parâmetros passados por **param**. Quando se faz *head* retorna o valor guardado no primeiro endereço do vetor.

6 Testes

No diretório de testes possui dois arquivos com erros semânticos e sintáticos **error.cipl** e **error1.cipl** e dois arquivos sem erros **success1.cipl** e **success2.cipl**.

Os erros semânticos no arquivos **error.cipl** é a expressão B de um construtor tem que ser do tipo int list ou float list ou **NIL** na linha 5 e coluna 9 e um *id* do tipo int sendo atribuído a um do tipo float list na linha 27 e coluna 9, os erros sintáticos são for() sem corpo linha 2 e coluna 8 e if sem corpo linha 33 e coluna 6. Além disso foi colocado dois erros léxicos nas linhas 18 e 24.

Os erros semânticos no arquivo **error1.c** é não declaração da *main* e uma chamada de função faltando um parâmetro na linha 24 e coluna 5. Os erros sintáticos são *filter* sem um parâmetro na linha 20 e coluna 8 e um *tail* sem um parâmetro na linha 22 coluna 5.

7 Instruções

Para se compilar o analisador sintático utiliza-se os seguintes comandos ou make:

```
bison ./src/syntax.y
--defines=./lib/syntax.tab.h -o ./src/syntax.tab.c; \
flex -o ./src/lex.yy.c ./src/lexico.l; \
cc -Wall -Wextra -Wpedantic -g ./src/*.c -o tradutor -Ilib -O0; \
./tradutor <caminho_do_arquivo_teste>
```

Referências

ALSU06. A. Aho, M. Lam, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2 edition, 2006.

Hec21. R. Heckendorn. C- grammar, 2021. `http://marvin.cs.uidaho.edu/Teaching/CS445/c-Grammar.pdf`. Acessado por último 02 Set 2021.

Lev09. J. R. Levine. *Flex and Bison*. O'reilly, 1 edition, 2009.

Mar21. J. D. Marangon. Compiladores para humanos, 2021. `https://johnidm.gitbooks.io/compiladores-para-humanos/content/`. Acessado por último 02 Set 2021.

Nal13. C. Nalon. Glc-precedencia, 2013. Notas de aula.

Pro12. Flex Project. Flex Manual, 2012. `https://westes.github.io/flex/manual/index.html`.Acessado por último em 30 Ago 2021.

RC21. R. Stallman R. Corbett. Bison, 2021. `https://www.gnu.org/software/bison/manual/bison.html`. Acessado por último 30 Ago 2021.

SP21. IME SP. Tabela de precendência c, 2021. `https://www.ime.usp.br/~yw/bcc2000/C/tabela.html`.Acessado por último 02 Set 2021.

A Gramática

$$
\begin{aligned}
\langle \text{program} \rangle &::= \langle \text{paramlist} \rangle && (1)\\
\langle \text{paramlist} \rangle &::= \langle \text{param} \rangle\ \langle \text{paramlist} \rangle \mid \langle \text{param} \rangle && (2)\\
\langle \text{param} \rangle &::= \langle \text{variableParam} \rangle \mid \langle \text{functionParam} \rangle && (3)\\
\langle \text{variableParam} \rangle &::= \texttt{TYPE LISTTYPE ID ;} && (4)\\
&\quad \mid \texttt{TYPE ID ;} && (5)\\
\langle \text{functionParam} \rangle &::= \texttt{TYPE ID (}\ \langle \text{functionParams} \rangle\ \texttt{)}\ \langle \text{stmt} \rangle && (6)\\
&\quad \mid \texttt{TYPE LISTTYPE ID (}\ \langle \text{functionParams} \rangle\ \texttt{)} && (7)\\
&\quad \langle \text{stmt} \rangle && (8)\\
\langle \text{functionParams} \rangle &::= \langle \text{functionParamsList} \rangle \mid \epsilon && (9)\\
\langle \text{functionParamsList} \rangle &::= \langle \text{functionParamsList} \rangle\ \texttt{, TYPE ID} && (10)\\
&\quad \mid \langle \text{functionParamsList} \rangle\ \texttt{, TYPE LISTTYPE} && (11)\\
&\quad \texttt{ID} && (12)\\
&\quad \mid \texttt{TYPE ID} && (13)\\
&\quad \mid \texttt{TYPE LISTTYPE ID} && (14)\\
\langle \text{call} \rangle &::= \texttt{ID (}\ \langle \text{argList} \rangle\ \texttt{)} && (15)\\
\langle \text{argList} \rangle &::= \langle \text{argList} \rangle\ \texttt{, ID} && (16)\\
&\quad \mid \texttt{ID} && (17)\\
&\quad \mid \epsilon && (18)\\
\langle \text{stmList} \rangle &::= \langle \text{stmList} \rangle \langle \text{stmt} \rangle \mid \langle \text{stmt} \rangle && (19)\\
\langle \text{stmt} \rangle &::= \langle \text{expStatement} \rangle \mid \langle \text{compoundStatement} \rangle && (20)\\
&\quad \mid \langle \text{ifStatement} \rangle \mid \langle \text{forStatement} \rangle && (21)
\end{aligned}
$$

| ⟨returnStatement⟩ | ⟨inputStatement⟩ (22)
| ⟨outputStatement⟩ (23)
| ⟨variableParam⟩ (24)

⟨expStatement⟩ ::= ⟨expression⟩ `;` (25)

⟨compoundStatement⟩ ::= `{` ⟨stmList⟩ `}` (26)
| `{ }` (27)

⟨ifStatement⟩ ::= `if (` ⟨expression⟩ `)` ⟨stmt⟩ (28)
| `if (` ⟨expression⟩ `)` ⟨stmt⟩ `else` ⟨stmt⟩ (29)

⟨forStatement⟩ ::= `for (` ⟨expStatement⟩⟨expStatement⟩ (30)
⟨expression⟩ `)`⟨stmt⟩ (31)

⟨returnStatement⟩ ::= `return` ⟨expression⟩ `;` (32)

⟨inputStatement⟩ ::= ⟨input⟩ `( ID ) ;` (33)

⟨outputStatement⟩ ::= ⟨output⟩ `(` ⟨term⟩ `) ;` (34)

⟨expression⟩ ::= `ID =` ⟨expression⟩ (35)
| ⟨orExpression⟩ (36)

⟨orExpression⟩ ::= ⟨orExpression⟩ `||` ⟨andExpression⟩ (37)

⟨andExpression⟩ ::= ⟨andExpression⟩ `&&` ⟨relationalExpression⟩ (38)
| ⟨relationalExpression⟩ (39)

⟨relationalExpression⟩ ::= ⟨relationalExpression⟩ ⟨REL_OP⟩ (40)
⟨listExpression⟩ (41)
| ⟨listExpression⟩ (42)

⟨listExpression⟩ ::= ⟨arithmExpression⟩ ⟨listOP⟩ (43)
⟨listExpression⟩ (44)
| ⟨arithmExpression⟩ (45)

⟨arithmExpression⟩ ::= ⟨arithmExpression⟩ ⟨SUB_ADD⟩ (46)
⟨airtmMulDivExpression⟩ (47)
| ⟨arihtmMulDivExpression⟩ (48)

⟨arithmMulDivExpression⟩ ::= ⟨arithmMulDivExpression⟩ ⟨MUL_DIV⟩ (49)
⟨term⟩ (50)
| ⟨term⟩ (51)

⟨term⟩ ::= ⟨const⟩ | ⟨call⟩ | `ID` (52)
| ⟨unaryTerm⟩ (53)
| ⟨immutable⟩ (54)

⟨unaryTerm⟩ ::= `!` ⟨term⟩ (55)
| `?` ⟨term⟩ (56)
| `%` ⟨term⟩ (57)

| ⟨SUB_ADD⟩ ⟨term⟩ (58)
⟨immutable⟩ ::= `(` ⟨expression⟩ `)` (59)
⟨const⟩ ::= `INT` | `FLOAT` | `STRING` | `NIL` (60)
⟨listOP⟩ ::= ⟨FUNCTION⟩ | ⟨INFIX⟩ (61)
⟨output⟩ ::= `writeln` | `write` (62)
⟨input⟩ ::= `read` (63)
⟨TYPE⟩ ::= `int` | `float` (64)
⟨LISTTYPE⟩ ::= `list` (65)
⟨REL_OP⟩ ::= `<` | `<=` | `==` | `>` | `>=` (66)
⟨INFIX⟩ ::= `:` (67)
⟨FUNCTION⟩ ::= `>>` | `<<` (68)
⟨SUB_ADD⟩ ::= `-` | `+` (69)
⟨MUL_DIV⟩ ::= `*` | `/` (70)
(71)

B Tabelas de Símbolos

Lexema	Expressão Regular e Descrição Informal	Exemplos
ID	qualquer letra com '_' seguido de um número [A-Za-z_]+[A-Za-z0-9_]*	jujuba_doce
LISTTYPE	qualquer tipo do list list	list
TYPES	tipos int e float int—float	int float
NULL	constante para listas vazias NIL	NIL
STRING	qualquer caracter dentro de \"[^\"]*\'[^']*	"chocolate"
ASSIGN	Um único caractere = =	=
IF	palavra chave if if	if
ELSE	palavra chave else else	else
FOR	palavra chave for for	for
RETURN	palavra chave return return	return
OUTPUT	Palavras da saída de dados da linguagem writeln — write	write writeln
INPUT	Palavra de entrada de dados read	read
INFIX	Palavras para construtores de lista :	:
FUNCTION	Caracteres para as função de map e filter >>— <<	>><<
COMMENT	Qualquer comentário em linha com // ou comentário multilinha com /**/ \/\/.*—\/*[^*\/]**\/	//pudim /* pudim é bom /
+ — - — * — /	Operações aritmeticas	+ - * /
—— — &&	Operações Lógicas	&& ——
>—<—>=—<=—==—!=	Operação de comparação	<<= >>= == !=
{}	Escopo	{}
()	Expressão	()
;	Separador de Expressão	;
?	Operação head da lista	?
!	Operação tail da lista ou negação	!
%	Operação tail destrutiva da lista	%
123	Número inteiro {DIGIT}+	123
.5	Número decimal {DIGIT}*\.{DIGIT}+(E[+—-]?{DIGIT}+)?	.5