Compiladores - Projecto

iJava

João Ricardo Lourenço, Nº 2011151194
Joaquim Pedro Bento Gonçalves Pratas Leitão, Nº 2011150072

2 de Junho de 2014

Relatório

Índice

1 Introdução

O presente trabalho pretende-se desenvolver um compilador para a linguagem $iJava$, um pequeno subconjunto da linguagem Java (versão 5.0). Por ser um subconjunto de uma outra linguagem, todos os programas que respeitem as regras impostas em $iJava$ são também, garantidamente, programas válidos em $Java$.

Nesta linguagem todos os programas são constituídos por uma única classe, que possui métodos e atributos estáticos, e públicos. Para além disso, a classe necessita obrigatoriamente de ter um método *main*, onde a execução do programa se inicia.

Podemos utilizar literais dos tipos inteiro e booleano e variáveis inteiras, booleanas e arrays uni-dimensionais de inteiros e booleanos.

A linguagem implementa também expressões aritméticas e lógicas, operações relacionais simples, instruções de atribuição e controlo ($if - else$ e $while$).

Os métodos definidos, e os respetivos valores de retorno, podem podem ser de qualquer tipo acima mencionado, com exceção do método *main*, que tal como em $Java$ possui como tipo de retorno o tipo *void*.

É também possível passar parâmetros (literais inteiros) ao nosso programa através da linha de comandos. É o método *main* que vai receber esses parâmetros, armazenando-os num array de objetos do tipo *String*. Embora este tipo de dados não esteja incluído na lista de tipos permitidos em $iJava$, a sua utilização apenas é permitida no método *main*, com a mera finalidade de obter os parâmetros passados ao programa aquando da sua invocação.

O desenvolvimento do compilador foi dividido em três fases distintas.

Numa primeira fase foi realizada a *Análise Lexical* do programa fonte, onde são identificados *tokens*, isto é, cadeias pertencentes à linguagem e que têm significado e relevância para o programa.

Seguiu-se a *Análise Semântica*, composta por quatro etapas principais: FIXME, MUDAR A MANEIRA COMO ESTAMOS A DIZER O QUE SE SEGUE

- **Tradução da gramática-fonte** (fornecida em notação $EBNF$) para o yacc 3 e realização da **Análise Sintática** do programa, permitindo assim reconhecer se as sequências de *tokens* que o constituem pertencem à linguagem, permitindo-nos assim detetar eventuais erros de sintaxe.

- **Construção da árvore de sintaxe abstrata**, etapa que é realizada em simultâneo com a *Análise Sintática*. A árvore de sintaxe abstrata irá representar o nosso programa a compilar, recorrendo a uma estrutura em árvore para representar as estruturas sintáticas das cadeias que constituem o programa a compilar.

- **Construção da tabela de símbolos**, utilizadas para armazenar informações relevantes sobre a classe (seus atributos e métodos), bem como sobre cada método definido pelo programador (como, por exemplo, o tipo de retorno e os argumentos).

- **Verificação de erros semânticos**, etapa principal da *Análise Semântica*, onde são realizadas verificações de tipos, garantindo que para cada operação a realizar não existem incompatibilidades de tipos entre os operandos nela envolvidos.

A última fase do trabalho consistiu na *Geração de Código Intermédio*, da qual resulta, na representação intermédia de $LLVM$, um programa equivalente ao que pretendemos compilar.

METER IMAGEM BONITINHA, TIPO CACEIRO???

2 Análise Lexical

Tal como referimos anteriormente, na *Análise Lexical* procedemos à identificação dos *tokens* da nossa linguagem. Para isso utilizámos a ferramenta *lex*, responsável por gerar analisadores lexicais para linguagens.

Assim, no nosso analisador, sempre que é detetada a presença de um comentário no programa a compilar, seja do tipo //... (comentários de apenas uma linha) ou do tipo $/*...*/$ (comentários multi-linha), os caracteres incluídos nesse comentário são ignorados.

Sempre que é detetado um caracter, ou uma sequência de caracteres, que não constitui nenhum *token* é detetado um erro lexical, sendo impressa uma mensagem de erro, indicando a existência de um caracter ilegal, juntamente com a sua posição no programa.

Adicionalmente, caso se verifique a ocorrência de um comentário multilinha que não foi devidamente terminado, o erro lexical é também detetado, sendo impressa uma mensagem de erro que indica a posição no programa onde o comentário foi iniciado.

2.1 Tokens

Em seguida, apresentamos a lista dos *tokens* válidos na linguagem $iJava$ e a lista dos *tokens* reservados que, por essa razão, não estão disponíveis na nossa linguagem:

- **ID**: Sequências alfanuméricas (maiúsculas e minúsculas) começadas por uma letra, podendo conter também símbolos como "_" e "$". Este *token* pode também ser descrito na forma da sua expressão regular: *letra(letra—[0-9])**, sendo o *token* **letra** da nossa autoria, definido por: $[a-z] \mid [A-Z] \mid$ "_"$"
- **INTLIT**: Sequências de dígitos decimais e hexadecimais (incluindo a-f e A-F) precedidas de $0x$. Este *token* pode também ser descrito na forma da seguinte expressão regular: *[0-9]+—0x[0-9a-fA-F]+*
- **BOOLLIT**: $true \mid false$
- **INT**: int
- **BOOL**: $boolean$
- **NEW**: new
- **IF**: if
- **ELSE**: $else$
- **WHILE**: $while$

- **PRINT**: *System.out.println*
- **PARSEINT**: *Integer.parseInt*
- **CLASS**: *class*
- **PUBLIC**: *public*
- **STATIC**: *static*
- **VOID**: *void*
- **STRING**: *String*
- **DOTLENGTH**: *.length*
- **RETURN**: *return*
- **OCURV**: (
- **CCURV**:)
- **OBRACE**: {
- **CBRACE**: }
- **OSQUARE**: [
- **CSQUARE**:]
- **OP1**: && |||
- **OP2**: <|>|==|! =|<=|>=
- **OP3**: " + " | −"
- **OP4**: " ∗ " | "/" | "%"
- **NOT**: "!"
- **ASSIGN**: " = "
- **SEMIC**: ";"
- **COMMA**: ","
- **RESERVED**: *abstract* | *continue* | *for* | *switch* | *assert* | *default* | *goto* | *package* | *synchronized* | *do* | *private* | *this* | *break* | *double* | *implements* | *protected* | *throw* | *byte* | *import* | *throws* | *case* | *enum* | *instanceof* | *transient* | *catch* | *extends* | *short* | *try* | *char* | *final* | *interface* | *finally* | *long* | *strictfp* | *volatile* | *const* | *float* | *native* | *super* | *null* | ++ | −−

Para além dos *tokens* apresentados, definimos outros *tokens*, que passamos a especificar:

- **NEWLINE**: *Token* correspondente ao caracter de mudança de linha, $\backslash n$
- **WHITESPACE**:*Token* correspondente ao caracter de espaço em branco
- **OPEN_COMMENT**: *Token* correspondente ao início de um comentário multi-linha, $/*$
- **CLOSE_COMMENT**: *Token* correspondente ao fecho de um comentário multi-linha, $*/$
- **SINGLE_LINE_COMMENT**: *Token* utilizado para detetar a ocorrência de um comentário de uma linha apenas

Quando implementámos a *Análise Sintática*, para resolver problemas de ambiguidade da gramática foi necessário, entre outras ações que iremos abordar na próxima secção, separar os *tokens* **OP1**, **OP2**, **OP3** e **OP4** nas diferentes sequências alfanuméricas que os constituíam. Assim, temos ainda os seguintes *tokens*:

- **AND** ("&&") e **OR** ("||"), originados a partir do *token* **OP1**
- **LE** ("≪"), **GE** ("≫"), **EQ** ("=="), **NEQ** ("!="), **LEQ** ("≪=") e **GEQ** ("≫="), originados a partir do *token* **OP2**
- **PLUS** ("+") e **MINUS** (-"), originados a partide do *token* **OP3**
- **MULT** ("*"), **DIV** ("/") e **MOD** ("%"), originados a partir do *token* **OP4**

2.2 Comentários

Para identificarmos a ocorrência de comentários nos programas a compilar recorremos aos *tokens* *OPEN_COMMENT*, *CLOSE_COMMENT* e *SINGLE_LINE_COMMENT*.

Quando detetamos o *token* *OPEN_COMMENT* é criado um novo estado no analisador, que indica a existência de um comentário multi-linha. A esse estamos demos o nome *MULTI_LINE_COMMENT_S*.

Uma vez neste estado, todos os caracteres e *tokens* identificados são ignorados, com exceção do *token* de fecho do comentário multi-linha (*CLOSE_COMMENT*) e do *token* de fim do ficheiro, $<< EOF >>$, disponível na ferramenta utilizada para desenvolver o analisador (*lex*).

Caso seja identificado o *token CLOSE_COMMENT*, o estado do analisador é reposto, passando este a ter o seu estado por defeito.

Por outro lado, se for detetado $<< EOF >>$ temos uma situação em que um comentário multi-linha não foi devidamente terminado, pelo que é gerado um erro lexical, terminando a execução do analisador e sendo o utilizador informado da ocorrência do erro e da localização no programa fornecido do comando que inicia o comentário.

Se, por alguma razão, o *token CLOSE_COMMENT* for identificado quando o analisador não se encontra no estado *MULTI_LINE_COMMENT_S* é detetada a ocorrência de um erro lexical, uma vez que na nossa linguagem não é possível a existência do comando */ sem que antes tenha sido colocado um /*. Mais uma vez, assim que o erro lexical é detetado o utilizador é informado com uma mensagem que indica a posição no programa onde se deu o erro, e o analisador termina a sua execução.

Ao detetarmos o *token SINGLE_LINE_COMMENT* vamos ignorar todos os caracteres e *tokens* que se lhe seguirem, até que seja reconhecido o *token* de mudança de linha (*NEWLINE*). Desta forma estamos a descartar toda a restante linha do programa, após a ocorrência de \\, tal como seria desejado proceder no tratamento de comentários de uma linha apenas.

2.3 Tratamento de Erros Lexicais

Tal como referimos nos pontos anteriores, sempre que o analisador desenvolvido deteta a ocorrência de um erro lexical (seja por existência de um *token* não permitido ou por não término de um comentário multi-linha), é impressa uma mensagem de erro que indica a posição do erro no programa a compilar (indicando a linha e coluna onde o erro ocorreu).

Quando é detetado um caracter ilegal, o analisador imprime a mensagem de erro, prosseguindo a sua execução na linha seguinte do programa a compilar até ser lido todo o conteúdo do programa.

Tendo sido detetada a ocorrência de um ou mais erros lexicais, após ler todo o programa, o analisador termina a sua execução, não sendo realizado mais nenhum passo da compilação.

3 Análise Sintática

A Análise Lexical permite-nos identificar os *tokens* da linguagem, isto é, os seus "átomos". No entanto, interessa-nos garantir que os *tokens* identificados estão organizados de acordo com a estrutura sintática da linguagem.

Para validar essa organização dos *tokens* necessitamos de utilizar uma gramática, que nos indica como devem estar organizados sintaticamente os *tokens* da linguagem. Para além disso, utilizámos a ferramenta *YACC (Yet Another Compiler Compiler)*, um gerador de analisadores sintáticos.

Assim, utilizamos o analisador lexical (*lex*) desenvolvido para reconhecer os *tokens* da linguagem, que são transferidos para o *yacc* onde, com base na gramática da linguagem fornecida, podemos verificar se o programa a compilar se encontra organizado de acordo com a estrutura sintática da linguagem.

No analisador sintático foi criada uma *union*, que define a estrutura de uma variável *yylval*, utilizada na comunicação entre o *yacc* e o *lex*. Apresentamos desde já a estrutura da *union* criada:

```
%union
{
    char* token;
    struct _node_t* node;
    int type;
}
```

Os *token* detetados no *lex* são comunicados ao analisador sintático através do campo *token* da variável *yylval*.

Para além disso, no analisador sintático definimos ainda três variáveis externas, para uso partilhado entre o *lex* e o *yacc*: Duas do tipo inteiro, *prev_col* e *prev_line*, e uma do tipo array de caracteres (*char**), chamada *yytext* e que contém a cadeia de caracteres lida pelo analisador, em cada momento.

Tal como o nome pode sugerir, *prev_col* e *prev_line* são utilizadas para armazenar o valor da coluna e linha (do programa a compilar) analisadas na iteração anterior da execução do analisador. A justificação para essa prende-se com as situações onde ocorrem erros sintáticos, sendo necessário informar o utilizador da localização no programa desse erro.

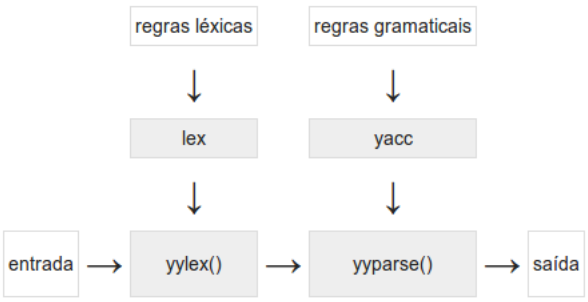

Figura 1: Ligação entre o *lex* e o *yacc*, ferramentas utilizadas para gerar os analisadores lexicais e sintáticos. Retirado de *http://pt.wikipedia.org/wiki/Yacc*

3.1 Gramática

De seguida, apresentamos a gramática da linguagem *iJava*, fornecida no enunciado do projeto, em notação **EBNF**:

Start → Program
Program → CLASS ID OBRACE FieldDecl | MethodDecl CBRACE
FieldDecl → STATIC VarDecl
MethodDecl → PUBLIC STATIC (Type | VOID) ID OCURV
[FormalParams] CCURV OBRACE VarDecl Statement CBRACE
FormalParams → Type ID COMMA Type ID
FormalParams → STRING OSQUARE CSQUARE ID
VarDecl → Type ID COMMA ID SEMIC
Type → (INT | BOOL) [OSQUARE CSQUARE]
Statement → OBRACE Statement CBRACE
Statement → IF OCURV Expr CCURV Statement [ELSE Statement]
Statement → WHILE OCURV Expr CCURV Statement
Statement → PRINT OCURV Expr CCURV SEMIC
Statement → ID [OSQUARE Expr CSQUARE] ASSIGN Expr SEMIC
Statement → RETURN [Expr] SEMIC
Expr → Expr (OP1 | OP2 | OP3 | OP4) Expr
Expr → Expr OSQUARE Expr CSQUARE
Expr → ID | INTLIT | BOOLLIT
Expr → NEW (INT | BOOL) OSQUARE Expr CSQUARE
Expr → OCURV Expr CCURV
Expr → Expr DOTLENGTH | (OP3 | NOT) Expr
Expr → PARSEINT OCURV ID OSQUARE Expr CSQUARE CCURV
Expr → ID OCURV [Args] CCURV
Args → Expr COMMA Expr

Lembramos que, em notação **ENBF**, os símbolos **[...]** englobam *tokens* opcionais e **{...}** implicam a repetição dos *tokens* 0 ou mais vezes.

3.1.1 Ambiguidade

Um análise mais cuidada da gramática apresentada permite-nos aferir da sua ambiguidade. Por exemplo, se num dado momento da análise sintática pretendermos analisar $2 + 3 * 5$, podemos reduzir esta expressão à variável *EXPR* da gramática por duas formas distintas:

1. Numa primeira abordagem podemos separar a expressão em **Expr** $\rightarrow$ **Expr + Expr** (onde o símbolo *"+"* é proveniente do *token OP3*). De seguida reduziríamos o primeiro símbolo *Expr* para o *token INTLIT* correspondente, neste caso, ao literal *"2"*. O segundo símbolo *Expr* seria desdobrado em **Expr** $\rightarrow$ **Expr * Expr** (onde o símbolo *"*"* é proveniente do *token OP4*). Neste caso os dois símbolos *Expr* seriam reduzidos ao *token INTLIT*, correspondente a cada um dos restantes literais.

2. Numa abordagem alternativa começaríamos por separar a expressão em **Expr** $\rightarrow$ **Expr * Expr**. De seguida desdobraríamos o primeiro símbolo *Expr* em **Expr** $\rightarrow$ **Expr + Expr**. Estes dois novos símbolos *Expr* seriam reduzidos a *INTLIT*, tal como o restante símbolo *Expr*.

Acabámos de provar a ambiguidade da gramática. Tal como esta situação existem muitas outras envolvendo os operadores englobados pelos *tokens OP1, OP2, OP3 e OP4*. Assim, é imperativa a definição de prioridades nos operadores, de forma a eliminar todas as ambiguidades presentes na gramática, permitindo assim a correta realização da análise sintática do programa.

3.1.2 Alterações à Gramática Fornecida

Uma vez que a gramática apresentada é ambígua e apresenta-se em notação **EBNF**, não aceite pela ferramenta utilizada para gerar o analisador sintático, necessitamos de alterar a gramática, de forma a eliminar as suas ambiguidades, tornando-a numa gramática aceite pela ferramenta utilizada.

Como já referimos, a primeira alteração realizada prende-se com a separação dos operadores englobados pelos *tokens OP1, OP2, OP3 e OP4*, criando um novo *token* para cada operador, definindo de seguida a prioridade dos diferentes operadores representados pelos *tokens* criados.

Utilizando a ferramenta *yacc* definimos do seguinte modo as prioridades dos operadores referidos:

```
%left OR
%left AND
```

```
%left EQ NEQ
%left LE GE LEQ GEQ
%left PLUS MINUS
%left MULT DIV MOD
%right ASSIGN
%left OBRACE
%right UNARY_HIGHEST_VAL
%left OSQUARE DOTLENGTH
```

FIXME: METER ALTERAÇÕES FEITAS À GRAMÁTICA. EXPLICAR QUAIS AS ALTERAÇÕES E PORQUE É QUE AS FIZEMOS

4 Construção da Árvore de Sintaxe Abstrata

FIXME: METER AQUI TAMBÉM O CÓDIGO: ESTRUTURA NODE_T E OS TIPOS E ENUM'S PRESENTES NO NODE_T.H

EXPLICAR AS ESTRUTURAS UTILIZADAS E O PORQUE DE TERMOS ESCOLHIDO ESTAS ESTRUTURAS. (PODERÍAMOS TER FEITO MELHOR? MUDARÍAMOS ALGUMA COISA?)

5 Análise Semântica

Até esta fase da compilação os passos executados, estruturas criadas, etc tinham como principal objetivo certificar que o programa, de facto, estava escrito na linguagem *iJava* adotada.

Na fase da *Análise Semântica* pretendemos estender esse estudo, procurando aferir se as instruções que constituem o programa possuem, de facto, algum significado em *iJava*.

Assim, nesta fase da compilação, a principal preocupação irá recair sobre as operações a realizar indicadas no programa, verificando a compatibilidade dos tipos de dados envolvidos nestas. Nesse sentido necessitamos de criar um estrutura que, de alguma forma, nos permita armazenar e consultar as variáveis (e os respetivos tipos) a que podemos aceder a partir de uma qualquer zona do programa.

Para tal, foi necessário criar e implementar *Tabelas de Símbolos*, que nos permitem aceder a essas mesmas informações. Desta forma podemos verificar os tipos dos dados envolvidos nas diversas operações que constituem o programa, detetando eventuais situações de incompatibilidade entre tipos, inválidas nos programas.

5.1 Tabelas de Símbolos

Uma *Tabela de Símbolos* (ou um *Ambiente*) é, como já referimos, uma estrutura de dados que mapeia identificadores com os seus tipos e localizações.

Como a nossa linguagem apenas permite a execução de programas com uma única classe, que possui métodos e atributos estáticos e públicos, necessitamos de conhecer todos os métodos e atributos que compõem a classe, bem como as variáveis definidas em cada um dos métodos.

Desta forma podemos facilmente detetar situações em que são invocados métodos não declarados na classe, ou em que são utilizados atributos não definidos na classe ou para o método no qual se pretende utilizar o atributo.

Por estas razões mantemos uma *Tabela de Símbolos* para a classe na qual está contido o programa, sendo também mantida uma *Tabela de Símbolos* para cada método definido para a classe.

5.1.1 Criação das Tabelas de Símbolos

Cada *Tabela de Símbolos* é criada a partir da *Árvore de Sintaxe Abstrata*, construída na etapa anterior da compilação.

Assim, para a criação da *Tabela de Símbolos* da classe do programa, percorremos os nós da *Árvore de Sintaxe Abstrata* relativos às declarações de atributos e métodos, criando uma nova entrada na tabela para cada método

ou atributo declarado, na qual se armazenam o nome do método ou atributo declarado.

No caso da entrada corresponder a um atributo é também armazenado o tipo do atributo. Caso a entrada corresponda à declaração de um método é guardado uma referência para a *Tabela de Símbolos* do método.

Na criação da *Tabela de Símbolos* de um método da classe percorremos os nós da *Árvore de Sintaxe Abstrata* relativos às instruções que compõem o método, criando na sua *Tabela de Símbolos* uma entrada onde armazenamos o seu nome, outra onde é armazenado o tipo de retorno, e uma entrada para cada argumento ou variável local declarada, contendo o respetivo tipo.

5.1.2 Implementação

Na listagem que se segue apresentamos a estrutura de dados utilizada para representar e implementar cada *Tabela de Símbolos* criada, à qual demos o nome de *sym_t*:

```
typedef struct _sym_t sym_t;

struct _sym_t{
        ijava_table_type_t node_type;
        char* id;
        ijavatype_t type;
        int is_parameter;
        sym_t* next;
        sym_t* table_method;
        node_t* method_start;
};

typedef enum {
        CLASS_TABLE,
        METHOD_TABLE,
        VARIABLE,
        METHOD
} ijava_table_type_t;
```

Chamamos também a atenção do leitor para a definição do tipo *ijava_table_type_t*, apresentado a par da estrutura *sym_t*.

Lembramos ainda que os tipos *ijavatype_t* e *node_t* foram apresentados na definição das estruturas utilizadas na *Árvore de Sintaxe Abstrata*.

Passemos a detalhar um pouco mais a implementação apresentada.

Cada entrada da *Tabela de Símbolos* corresponde a uma estrutura do tipo *sym_t*, sendo cada tabela constituída por uma ou mais entradas.

As nossas *Tabelas de Símbolos* não passam de simples listas ligadas, onde o primeiro elemento (a cabeça da lista) caracteriza a tabela, possuindo

informação relativa ao seu nome (que coincide com o nome da classe ou do método em questão), armazenado em *id*, e tipo, que se encontra em *node_type*. O tipo indica-nos se a entrada em questão corresponde a uma tabela de uma classe ou de um método, ou a uma declaração de um método ou de uma variável. Os diferentes elementos da lista encontram-se ligados pelo valor armazenado em *next*, um ponteiro para o próximo elemento da lista.

No caso da *Tabela de Símbolos* da classe, os restantes elementos vão corresponder a declarações de atributos ou métodos, possuindo no seu campo *id* o nome do atributo ou método declarado. Os que correspondem a declarações de atributos da classe vão possuir o seu tipo (*int*, *boolean*, etc) armazenado em *type*. Já os que correspondem a declarações de métodos referenciam a *Tabela de Símbolos* desse método através do campo *table_method*. Adicionalmente, estes últimos possuem também no campo *method_start* uma referência para o nó da *Árvore de Sintaxe Abstrata* onde se encontra a declaração do método.

Relativamente à *Tabela de Símbolos* de um método, o seu primeiro elemento está de acordo com o descrito anteriormente. Segue-se um elemento que contém o tipo de retorno do método, armazenado no campo *type*. Posteriormente encontram-se os argumentos do método, no caso de existirem, também eles com a informação relativa ao nome e tipo presente em *id* e *type*, respetivamente. Estes elementos possuem também o valor 1 no campo inteiro *is_parameter*, que indica que se tratam de parâmetros do método. Por fim, encontram-se as variáveis locais, pela ordem de declaração no método, sendo a informação relativa ao seu nome e tipo armazenada nos campos *id* e *type*, respetivamente.

5.1.3 Conclusão

Após a análise das estruturas de dados apresentadas podemos retirar algumas conclusões acerca das implicações que a escolha destas estruturas teve na implementação das *Tabelas de Símbolos*.

Em primeiro lugar é clara a utilização da mesma representação para estruturas conceptualmente distintas, como é o caso da tabela de símbolos da classe e dos métodos, ou das declarações de atributos e métodos de uma classe.

De facto, reconhecemos que a utilização de diferentes estruturas seria mais adequado de um ponto de vista conceptual, já que nos permitiria distinguir melhor os elementos que pretendemos representar, evitando a existência de campos não utilizados em algumas estruturas.

No entanto, optámos por manter esta representação dado que a consideramos mais simples ao nível da implementação, permitindo-nos generalizar algumas operações como a impressão dos elementos das diferentes tabelas, e até a procura de um elemento numa tabela.

5.2 Deteção de Erros Semânticos

6 Geração de Código

7 Apreciação do Trabalho